UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Virtualización mediante tecnología SR-IOV de tarjetas de red de altas prestaciones basadas en lógica programable

Autor: José Fernando Zazo Rollón,
Tutor: Dr. Sergio López Buedo

SEPTIEMBRE 2015

VIRTUALIZACIÓN MEDIANTE TECNOLOGÍA SR-IOV DE TARJETAS DE RED DE ALTAS PRESTACIONES BASADAS EN LÓGICA PROGRAMABLE

AUTOR: José Fernando Zazo Rollón
TUTOR: Dr. Sergio López Buedo

High Performance Computing and Networking Research Group
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
SEPTIEMBRE 2015

Abstract

***Abstract* —** The IT industry has been taking the most demanding and rigorous standards as the reference in order to achieve stability, a high fidelity to protocols and a proper quality of the final product. Whilst this model may have been useful for the past, it is inevitable that time to market becomes a crucial bottleneck when developing custom hardware for network appliances. At this point, Network Function Virtualization (NFV) allows creating specialized solutions with general-purpose equipment. Broadly speaking, computing is transferred from the hardware layer to a CPU-based software.

The main objective to treat is the exploration of the use of FPGAs, and its connectivity with the host system, as a feasible replacement for traditional hardware (switches, routers, etc.) in multigigabit networking environment. Own developments are disclosure under a free license as well as the underlying technologies are conscientiously tracked. From a DMA engine capable of ensuring the data transmission with rates above 40 gigabit per second (with measured peaks of over 50 Gbps), to the device controllers needed to interact with the system, are explained to the reader. The final reference platform consists of a network interface card (NIC) which involves as many virtual functions (VFs) as instantiated interfaces. The transmitted/received information by every abstract device is processed individually, in a transparent way to the developer, with destination/source the computer network. The key concept is known as SR-IOV, which accompanies by a FPGA, eases the virtualization of multiple functionalities. Independently, several instances of virtual machines may access to a VF exclusively thanks to PCI passthrough capabilities.

The independency of the host station hardware, and the flexibility of the suggested framework, assure the user a notorious trade-off between performance and time production. The popular believing that high performance computing is confronted with virtualization heads to a wrong conclusion. In particular, an environment where the data is processed at 40 Gbps has been released. However, the subjacent virtualization support by the hardware platform (IOMMU) is limited and, in the system to card direction, the transferences suffer a pronounced bottleneck (10% of the performance of the native experiments) whilst this effect is palliated in the card to system direction (over 90% of the native results).

***Index Terms* —** Network Function Virtualization, Virtual Network Appliance, FPGA-based acceleration, SR-IOV, PCI Passthrough, PCIe, DMA engine

Resumen

***Resumen* —** La industria de las telecomunicaciones ha seguido estándares muy rigurosos que aseguren la estabilidad, fidelidad al protocolo y calidad de los productos desarrollados. Mientras que este modelo ha funcionado bien en el pasado, son inevitables unos ciclos de producción largos con un lento avance en el hardware especializado. Es en este punto, donde la virtualización permite generar equipos especializados con elementos de propósito general. Se traspasa parte de la computación desde un elemento puramente dedicado a la CPU del sistema (virtualización de funciones de red, NFV) concediendo una gran dinamicidad al entorno.

El objetivo primordial es la exploración de la viabilidad del uso de FPGAs y la conectividad con el sistema anfitrión (basado en software) como sustituto para el hardware tradicional (*switches*, *routers*, etc.) en entornos multigigabit. Los desarrollos propios son liberados como contribuciones de licencia libre y las tecnologías subyacentes estudiadas en amplio detalle. Se implementa desde un motor de DMA que permita asegurar una tasa de transferencia sostenida para enlaces de 40 gigabits por segundo (mediciones tomadas por encima de 50 Gbps), hasta los controladores necesarios para la interacción con el dispositivo. La plataforma final de referencia consiste en una tarjeta de red con tantas funciones virtuales como interfaces existan. La información transmitida/recibida por cada dispositivo abstracto es tratada de manera independiente, transparente al desarrollador, con destino/origen final/inicial la red de ordenadores. La tecnología clave presentada para este proceso se conoce como SR-IOV, que acompañada por una única placa FPGA, facilita la simulación de múltiples periféricos dedicados. De manera independiente, distintas instancias de máquinas virtuales son capaces de hacer un uso exclusivo del dispositivo gracias a las capacidades de PCI *passthrough*, ofreciendo la falsa sensación de disponer de un recurso para su explotación individual. La independencia de la estación anfitriona, en cuanto a configuración hardware se refiere, y la marcada flexibilidad de los diseños, favorecen que esta arquitectura ofrezca un buen compromiso entre rendimiento y tiempo de puesta en mercado. Se desmiente la falsa creencia de que virtualización está reñida con procesamiento de alto rendimiento en todos los escenarios, aunque se han localizado carencias en el soporte por parte del hardware actual. En particular, la cantidad máxima de datos transferibles se ve limitada y aplicaciones que hagan uso intensivo en las comunicaciones hacia la tarjeta pueden verse gravemente afectadas (10% del rendimiento total en las pruebas generadas) si hacen uso de la virtualización. En la dirección inversa, un rendimiento superior al 90% ha sido probado.

***Palabras Clave* —** Virtualización de funciones de red, aceleración basada en FPGA, SR-IOV, PCI Passthrough, PCIe, motor DMA

Mi más sincero reconocimiento a todos los miembros del grupo de investigación HPCN ya que sin su ayuda y medios este proyecto no hubiera sido posible. Gracias también al equipo técnico de Naudit cuya experiencia y valía en el sector se han hecho notar.

Agradecer, por último, a todos esos amigos y familiares con los que siempre se ha podido contar.

Índice general

Índice de tablas

Índice de figuras

Índice de términos

ARI Interpretación alternativa del identificador de rutado, del inglés *Alternate Routing ID*. 39, 65, 74, 75

BAR Región de memoria configurable en la FPGA por el sistema, del inglés *base address register*. 39, 64, 66, 72, 74

BDF Función de dispositivo de bus, del inglés *Bus Device Function*. 62, 63, 65, 74, 75

C2S Abreviación para transferencias desde la tarjeta FPGA al sistema anfitrión, del inglés *Card to System*. 34, 35, 48–51

CPD Centro de procesamiento de datos. 54

CPU Unidad central de procesamiento, del inglés *central processing unit*. 4, 10, 11, 13, 14, 17, 20, 25, 28, 29, 34, 39, 44, 46–48, 52, 54, 55, 61, 67, 68, 70, 75

DMA El acceso directo a memoria permite acceder a la memoria del sistema para leer o escribir independientemente de la unidad central de procesamiento principal. 4, 11, 12, 25, 27, 34, 35, 39, 40, 42–44, 46–49, 52

FIFO Propiedad de ciertas estructuras de datos, consistente en que el primer dato insertado coincide con el primero en salir. 25, 27, 35, 37

FPGA Dispositivo hardware programable, del inglés *Field Programmable Gate Array*. 1, 2, 4, 10–12, 14–17, 20, 22–24, 27–29, 36, 39, 43, 46–48, 51–53

Gbps Unidad de transferencia, gigabits (10^9 bits) transferidos en un segundo. 2, 4, 10, 18, 24, 32, 46, 47, 49, 50, 52

GPU Unidad de procesamiento gráfico. 10, 59, 60

HDL Lenguaje de descripción hardware, del inglés *Hardware description language*. 17, 20, 24, 36, 47

IO Abreviación para entrada/salida. 55–57, 60–63, 66, 69

IOCTL Es una llamada de sistema en Unix que permite a una aplicación controlar o comunicarse con el driver de un dispositivo. 43, 44

IOMMU IO Memory Management Unit. 61

IoT Internet de las cosas, del inglés *Internet of Things*. 1, 6

IP core Bloque lógico de datos. 24, 25, 27, 28, 31, 34, 46, 47, 75, 76

KVM Máquinas virtuales basadas en el kernel de Linux, del inglés *kernel-based virtual machine*. 48, 55, 58–60

LXC Contenedor de Linux, del inglés *Linux Container*. 59, 60

MAC En redes de telecomunicaciones se trata de una subcapa del nivel de enlace en el modelo OSI. 25, 46

MDIO Abreviación del inglés *Management Data Input/Output*. Es un bus definido en la especificación Ethernet 802.3 para el control de la interfaz independiente del medio, MII.. 25

MMU Memory Management Unit. 44

MSI Método de generación de señales a través del uso de mensajes. 34, 69, 71–73

MSI-X Extensión del protocolo definido por MSI. 34, 35, 39, 40, 52, 71–73

NFV Virtualización de funciones de red, del inglés *Network Functions Virtualization*. 1, 2, 5–9, 12, 14, 17, 22

NIC Tarjeta de red, del inglés *Network Interface Card*. 1, 2, 16, 17

OSI Modelo de red descriptivo, creado por la Organización Internacional para la Estandarización (ISO) en el año 1980. 24

PCI Bus para la interconexión de periféricos, del inglés *Peripheral Component Interconnect*. 2, 62, 63, 68, 69, 71, 73

PCIe Bus para la interconexión de periféricos ofreciendo altas prestaciones. 2, 4, 10, 11, 19, 24, 25, 27–30, 32, 34, 37, 42, 49, 62–64, 69

PF Función física. 37, 41, 42, 48, 62, 65, 66

PIC Controlador programable de interrupciones, del inglés *Programmable Interrupt Controller*. 67, 68

RAM Memoria de acceso aleatorio, del inglés *Random Access Memory*. 47

S2C Abreviación para transferencias desde el sistema anfitrión a la tarjeta FPGA, del inglés *System to Card*. 34, 35, 48–51

SDN Red definida en software, del inglés *Software Defined Networking*. 1, 2, 6–9, 17, 22

SR-IOV Virtualización de entrada salida de un único nodo, del inglés *Single Root IO Virtual*. 3–5, 13–16, 18, 19, 23, 27, 37, 48, 50–53, 62, 64–66

TCP Protocolo de comunicación orientado a conexión fiable del nivel de transporte. 24

TLP Acrónimo para Transaction Layer Packet o paquete de la capa de transacción en el bus PCI. 25, 27–30, 32, 36, 37, 47, 49

VF Función virtual. 27, 35, 37, 39–42, 48, 50, 62, 64–66, 74

VM Máquina virtual, del inglés *virtual machine*. 3, 13–16, 18, 41, 44, 48, 50, 52–54, 57, 59–62

VMM Monitor de máquinas virtuales, del inglés *virtual machine monitor*. 54, 56–58, 60, 61

VT-d Virtualization Technology for Directed IO. 61

XAUI Estándar para extender el protocolo XGMII entre las capas MAC y PHY en 10 Gigabit Ethernet. 47

1 Introducción

1.1 Motivación

Tras décadas manteniendo una configuración muy similar en las redes de computadores, el gran aumento en el número de dispositivos en el internet de las cosas (IoT) está acrecentando la necesidad de modificar el enfoque clásico. Y es que los cambios en la industria han sido muy acentuados, se han dejado apartadas las tendencias tradicionales para generar una gran avalancha de aplicaciones, la computación en la nube y el almacenamiento y/o procesamiento de grandes volúmenes de datos (*big data*).

Las redes telemáticas deben enfrentarse a toda una serie de retos no planteados inicialmente. Desde el nuevo e incrementado número de usuarios hasta la adaptación para los nuevos servicios y sus exigencias. La reducción de costes, así como la simplificación en la gestión de los elementos de red, han derivado en dos conceptos claves en el día de hoy. Son las redes definidas en software (SDNs) y la virtualización de funciones de red (NFV). Ambos enfoques, aunque complementarios, persiguen un mismo objetivo: flexibilidad y agilidad para el diseño y gestión de las infraestructuras para la comunicación.

Mientras que SDN persigue la separación de los elementos de control de la red (software) de la comunicación de datos (hardware), NFV permite implementar parte de la funcionalidad requerida para el tratamiento de información en software. La principal diferencia radica en que no necesariamente las características software en el caso de NFV están ligadas a la capa de control. Como ejemplo, la funcionalidad de cifrado del tráfico, en lugar de ser computada directamente en hardware, su ejecución es viable de ser aplicada a nivel de servidor, router, switch, etc., abstrayendo de las competencias de más bajo nivel.

El hardware reconfigurable es un candidato razonable para el reemplazo de las tarjetas de red (NICs) convencionales, garantizando hasta cierto punto la eficiencia de un desarrollo hardware puro pero con la flexibilidad del software para adaptar la funcionalidad a la cambiante demanda sin la sustitución de la plataforma. Un refinamiento a la aplicación de dispositivos hardware programables (FPGAs) de manera independiente, que permita exprimir los beneficios de SDN y NFV, es la interacción de la misma con una estación hardware anfitriona.

Aunque se trata de un diseño conceptualmente comprensible, los desarrollos libres que permitan aprovechar esta tecnología son muy limitados y, en la gran mayoría de las ocasiones, de código cerrado derivando en un menor impacto en entornos de producción.

1.2 Objetivos

El principal objetivo es la evaluación de la idoneidad de la sustitución de NICs de altas prestaciones por FPGAs en entornos donde se persigue una alta abstracción de la plataforma subyacente tanto a nivel lógico (sistema operativo) como físico (hardware).

Tanto la investigación como estudio de las tecnologías involucradas es la principal cuestión a abordar a lo largo del resto de documento. A su vez, se intercalan los aspectos más teóricos con la propuesta y evaluación empírica de posibles maneras de abordar el problema. En líneas generales cuatro son los grandes puntos a indagar:

(a) Creación de un diseño para hardware reconfigurable que cumpla las necesidades de una NIC. Comunicación satisfactoria a tasa de línea en redes *ethernet* multigigabit de 10 gigabit por segundo (Gbps).

(b) Soporte para la comunicación entre periférico y sistema operativo anfitrión. Esto requiere tanto de un soporte por parte de la FPGA como de módulos controladores que permitan manejar el dispositivo. La interacción con estos módulos debe mantenerse accesible y sencilla de modo que la utilización por terceros no requiera de conocimientos intrínsecos del hardware.

(c) La virtualización de las funciones del dispositivo, su anexión a una máquina virtual y la correcta transparencia entre la información vertida por distintas fuentes es el último punto con el que se completa el entorno.

(d) Desde un punto de vista más global, el fin último es el planteamiento de una arquitectura escalable y flexible que garantice la consecución de los hitos perseguidos por las tecnologías SDN y NFV en entornos más variopintos.

1.3 Retos

Una de las dificultades principales en las aproximaciones aceleradas mediante FPGAs es la comunicación entre las aplicaciones corriendo en la arquitectura anfitriona y la unidad de coprocesamiento. Esta complejidad se acentúa si múltiples estaciones intentan acceder a un mismo recurso. Por fortuna, al compartir recursos comunes tanto NICs como FPGAs, como la conexión al sistema anfitrión mediante el bus para la interconexión de periféricos (PCI) express (PCIe), las mismas técnicas que garantizan el éxito en entornos con soporte para la virtualización son aplicables.

El acto de crear un elemento abstracto (no real) de un recurso, es logrado con la tecnología de virtualización de entrada/salida de un único nodo (SR-IOV). Gracias a ello, una máquina virtual (VM) es capaz de observar un periférico como suyo propio. En conjunción con *PCI passthrough* se favorece una menor penalización en el rendimiento causada por la utilización de VMs. El precio a pagar por la independencia que se obtiene tanto de los medios físicos de la máquina anfitriona como de la capa software es este pequeño *overhead* que, aunque se ha hecho un gran esfuerzo en su paliación, la creencia popular sigue sustentándose en la premisa de que la virtualización esta reñida con la computación de alto rendimiento.

No obstante, debido a la falta de patrones claramente especificados para la virtualización de dispositivos, la sincronización entre los múltiples procesos que acceden concurrentemente a un mismo recurso real, a través de las funciones virtuales que provee, no sigue un estándar. Es el propio desarrollador el que debe asegurar los mecanismos de sincronismo en la capa que éste desee. Es decir, tanto una sincronización a nivel de un módulo controlador como que el mismo hardware sea el encargado de arbitrar el acceso, son opciones viables. En ambos casos, se requieren de conocimientos profundos de la plataforma, ya sea a la hora de desarrollar mecanismos de comunicación entre *drivers* instanciados en la máquina virtual y la anfitriona o, por el contrario, del despliegue de los recursos oportunos para la gestión en la plataforma física.

En último lugar, pero no por ello menos importante, se está planteando una arquitectura innovadora que incorpore desarrollos de muy bajo nivel. Un diseño hardware corriendo a 250MHz o, la comunicación de datos entre controladores y programas de usuario que no requieran de copias intermedias, son particularidades a las que se debe hacer frente a lo largo de todo el proceso. Todas las demoras innatas a un desarrollo de estas características deben ser salvadas, de modo que su reutilización e implantación en un futuro por parte de terceros sea una tarea más inmediata.

1.4 Metodología

En un proyecto tecnológico es necesario prefijar y definir de manera unívoca tanto los hitos como los aspectos a tratar en cada una de las etapas de desarrollo para asegurar la correcta evolución del proyecto a medida que se van produciendo los primeros avances. Con tal fin se establece un modelo de desarrollo incremental e iterativo, donde inicialmente se plantea una arquitectura a nivel conceptual. Ésta será ejemplificada como una tarjeta de red implementada en hardware reconfigurable que figura como múltiples dispositivos virtuales desde el punto de vista de usuario. En toda esta de etapa de desarrollo de un producto software/hardware, se definen una serie de metas intermedias, que colaboren en la obtención del producto final. Se contemplan los siguientes hitos:

(a) Desarrollo de módulo controlador del dispositivo y programas de usuario encargados de la interacción. En primer lugar se persigue la generación de un *driver* propio con funcionalidades básicas para la comunicación con una FPGA. Tanto comunicación mediante acceso directo a memoria (DMA) como por acción directa de la unidad central de procesamiento (CPU) debe ser soportada. Dada la carencia de una plataforma hardware inicial, se toman los diseños de referencia de la placa objetivo como diseño auxiliar que provea de la asistencia para las comunicaciones a nivel de periférico.

(b) Desarrollo de tarjeta de red convencional en hardware reconfigurable. Un primer diseño hardware que asegure la comunicación de los datos por PCIe hacia/desde la tarjeta por parte del diseño. Niveles de abstracción superiores a la capa física no deben ser desarrollados.

(c) Sustitución de una alternativa comercial para la transferencia de datos DMA aplicada en el hito anterior por una solución libre. Aunque un rendimiento elevado sería recomendable, como aproximación inicial no es esencial.

(d) Dotación al *endpoint* de PCIe del soporte para SR-IOV. Los módulos controladores deben ser adaptados a las nuevas exigencias y el diseño hardware, más allá de este componente, si así se requiriese.

(e) Refinamiento del diseño de referencia integrado para su adecuación a entornos multigigabit con tasas de hasta 40 Gbps agregada.

Para la culminación éxitosa de los elementos citados las tecnologías seleccionadas deben permitir el tratamiento de los datos de manera precisa y eficiente. Podría pensarse en el empleo de lenguajes de alto nivel para la realización de los programas a nivel de usuario. Sin embargo, al primar el rendimiento, todos los desarrollos software se codifican en lenguaje C/C++, mientras que a nivel de desarrollo hardware se emplea Verilog. Esta última decisión, en contraposición a la utilización de herramientas de síntesis desde lenguajes de alto nivel, está motivada por el marcado temporal preciso que se consigue al trabajar a nivel de registro. C/C++, por el contrario, ha sido seleccionado por la gran cantidad de funcionalidades aportadas a la hora de interactuar con el hardware subyacente, desde la especificación de la afinidad a los procesadores en procesos software hasta la comunicación con módulos controladores o el manejo eficiente de la memoria (si así ha sido implementado el programa).

De cara a la comprobación del entorno, tres son los puntos para la correcta verificación: depuración a nivel lógico (uso de *testbenchs* que modelen el comportamiento de la arquitectura hardware o de depuradores para las aplicaciones software), depuración en entorno real (mediante el aprovechamiento de los recursos libres de la FPGA para la monitorización de señales físicas) y, finalmente, validación de los resultados obtenidos en un entorno real.

1.5 Estructura del documento

Como toma de contacto se comienza presentando una arquitectura híbrida entre hardware y software que permita el tratamiento de la información en enlaces de red multigigabit en la Sección 2. En la Sección 3 se realiza una incursión en herramientas previamente desarrolladas para la aceleración de aplicaciones NFV a la par que se detalla cómo otras soluciones han sabido enfrentarse de manera individual a los problemas tanto de transmisión de datos como de virtualización de funciones.

La implementación del entorno propuesto por el estudiante para la resolución del problema se localiza en la Sección 4 mientras que los resultados tanto teóricos como empíricos figuran en la Sección 5. Cabe destacar que todos los desarrollos se pueden adquirir bajo una licencia libre. Se concluye el documento en la Sección 6 informando de las conclusiones finales y el trabajo a realizar. A lo largo del Apéndice A se detallan las tecnologías involucradas en el proceso: hipervisores de máquinas virtuales junto a los distintos paradigmas, información detallada sobre *PCI passthrough*, SR-IOV y cómo las interrupciones son gestionadas en la actualidad. En otras palabras, en este punto se agrupan todas las tecnologías que deben contar con soporte tanto del hardware subyacente como del sistema operativo anfitrión. Aunque esenciales para el propósito del trabajo, representa un área del conocimiento menos ligado a la línea general de la generación de un entorno virtualizado con competencias para el tratamiento por el software de los datos.

2 Arquitectura propuesta

2.1 SDN y NFV

El internet de las cosas ha causado una gran revolución en cómo las redes y las infraestructuras de telecomunicaciones son gestionadas. El IoT es un mercado prometedor, una gran fuente a explotar por la industria y donde el número de dispositivos conectados a la red se espera que siga aumentando durante las próximas décadas.

Este ecosistema, adicionalmente, acarrea la implantación de nuevos servicios que originariamente eran desconocidos, abarcando desde la implantación en la industria manufacturera (incluyendo producción, distribución y seguimiento), la salud (asistencia doméstica, bienestar del individuo), transporte, administración, seguridad pública (vigilancia), mercado automovilístico, agricultura hasta la participación activa de la población. Millares de aplicaciones están impactando en este mercado y la tendencia parece implicar nuevos objetivos. Por ejemplo, reconocimiento biométrico de los individuos o la robótica (en campos como la domótica) que tendrán un significativo impacto en las redes de comunicaciones.

A pesar de conocer esta moda, su comportamiento fluctuable y no predecible no permite anticiparse en gran medida a los acontecimientos venideros. Cada vez la comunicación involucra más elementos automatizados (detrás de las comunicaciones ya no siempre se localiza un usuario físico, comunicación máquina-máquina) y sus requerimientos varían enormemente.

Un nuevo entorno de desarrollo está pendiente de madurar. Debe cumplir con los requerimientos de rendimiento y disponibilidad pero de diseño ágil. La nueva infraestructura propuesta en este trabajo explota los conceptos de la virtualización, en el sentido de que permite la adición de nuevas extensiones a la plataforma actual con un coste idealmente ínfimo y con una abstracción de la plataforma bastante elevada. Sin embargo, antes de entrar en detalles, es preciso mencionar las líneas de trabajo actuales, donde destacan SDN y NFV sobre el resto de las alternativas.

2.1.1 SDN

SDN tiene como objetivo simplificar la programación de funciones de red mediante el uso de software. Con origen en un ambiente investigador, tres son los pilares fundamentales en los que se sostiene: desglose de los distintos planos en la transferencia de información, procesamiento mediante hardware diferenciado y dinamicidad de las operaciones.

(a) Control y manejo de datos desglosado. Mientras que en las redes convencionales tanto el plano de control (incluyendo el de configuración) y datos está implementado en *routers* y *switches*, SDN elimina la potestad de manejar el plano de control al hardware.

(b) La lógica de control de los datos es desplazada a una entidad externa. Recibe el nombre de controlador SDN o sistema operativo de red.

(c) El comportamiento de la red es programable a través de aplicaciones software que corren por encima del sistema operativo de red y que interactúan con el plano de control.

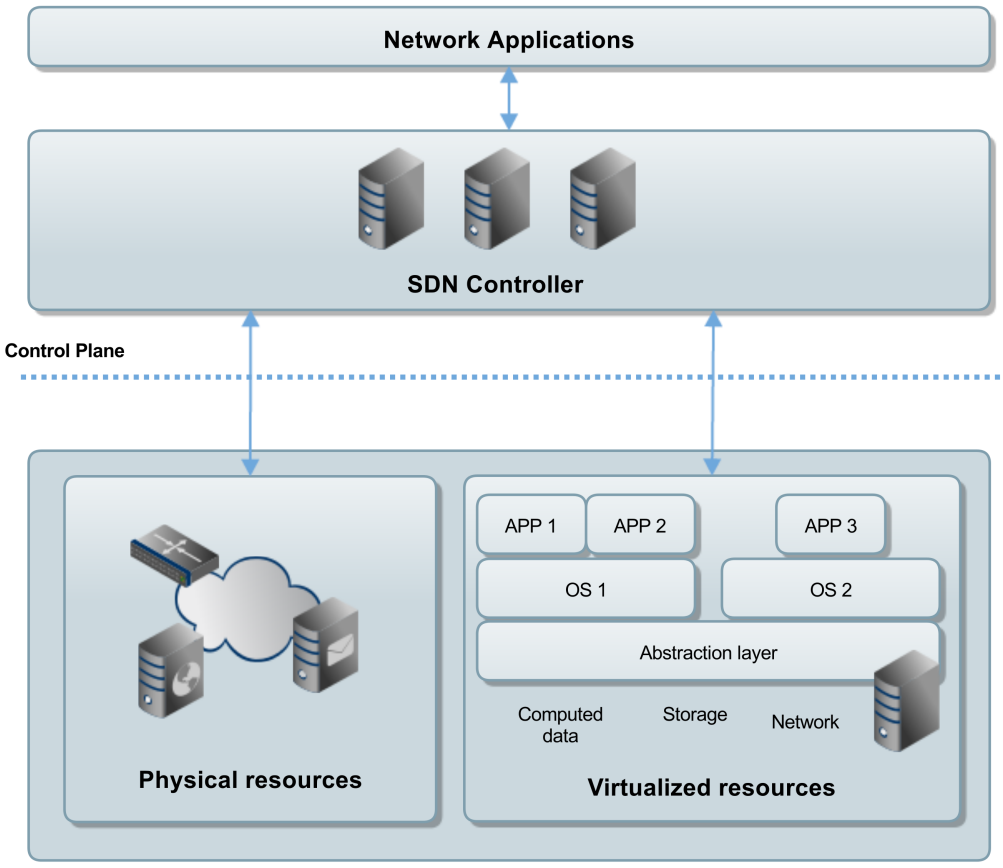


Figura 2.1: Esquema de SDN.

Como se muestra en la Figura 2.1, el controlador de SDN tiene dos conexiones principales: una interfaz hacia los recursos físicos o virtuales (por ejemplo, si se usa junto a NFV) y otra conexión hacia el gestor de aplicaciones que pudiera (o no) ser ejecutado sobre la misma plataforma hardware que el controlador. Ejemplos de protocolos para la intercomunicación mediante estas interfaces son *OpenFlow* para la interacción con los recursos y *REST APIs* para la comunicación con el gestor de aplicaciones.

Aunque a primera vista las aplicaciones más complejas queden enmascaradas por la eficiencia en términos de coste de producción o por los nuevos productos más escalables o mantenibles, existe una gran ventaja en el momento de distribuir los recursos de manera dinámica. Desde la aplicación de reglas de rutado de manera semiautomática (en relación con el comportamiento del tráfico observado) para evitar la sobrecarga de nodos particulares hasta el redireccionamiento concreto de protocolos a estaciones particulares. Por ejemplo, la prioritización de protocolos mediante su direccionamiento a través de estaciones más potentes o menos saturadas puede ser aplicado en tiempo real y sin modificaciones ni en los equipos hardware ni en los protocolos. La figura del administrador de red queda descargada de trabajo y se asegura una autonomía independiente de determinadas funcionalidades.

2.1.2 NFV

Las funciones de red en entornos no virtualizados son meramente una solución específica de un vendedor que involucra hardware y software, comúnmente referida como nodo o elemento de red. La virtualización de las funciones de red introduce un avance adicional, al abstraer el software del hardware.

Los elementos de la red no son nunca más un bloque rígido donde el software depende de la estación anfitriona. Esto favorece que ambos entornos, de manera independiente, puedan seguir prosperando y mejorando sin necesidad (o con la mínima adaptación) del otro componente.

En la especificación de NFV [ETSI, 2013], consultar Figura 2.2, tres son los dominios claramente identificados: función de red virtualizada, infraestructura y administración del entorno.

- La función de red virtualizada es aquella pieza de software que implementa una tarea determinada donde se involucra un procesamiento de los datos de la red.
- Infraestructura NFV. Plataforma subyacente capaz de ejecutar funciones virtualizadas, ofreciendo una abstracción del hardware real del equipo.
- Árbitro de NFV que se encarga de todos los aspectos asociados al manejo de las tareas virtualizadas en el entorno. Coordina el despliegue de nuevos recursos tomando en consideración varios criterios como rendimiento o afinidad.

Nótese cierta analogía en las características con SDN. Las funcionalidades implementadas a nivel de software están completamente abstraídas de la plataforma en niveles inferiores aunque su aplicación es ligeramente diferente. Mientras que SDN se limitaba al plano de control (y mantenimiento como subconjunto de éste), NFV ofrece una plataforma genérica para el tratamiento de los datos de la red por parte del software. De este modo, una implementación

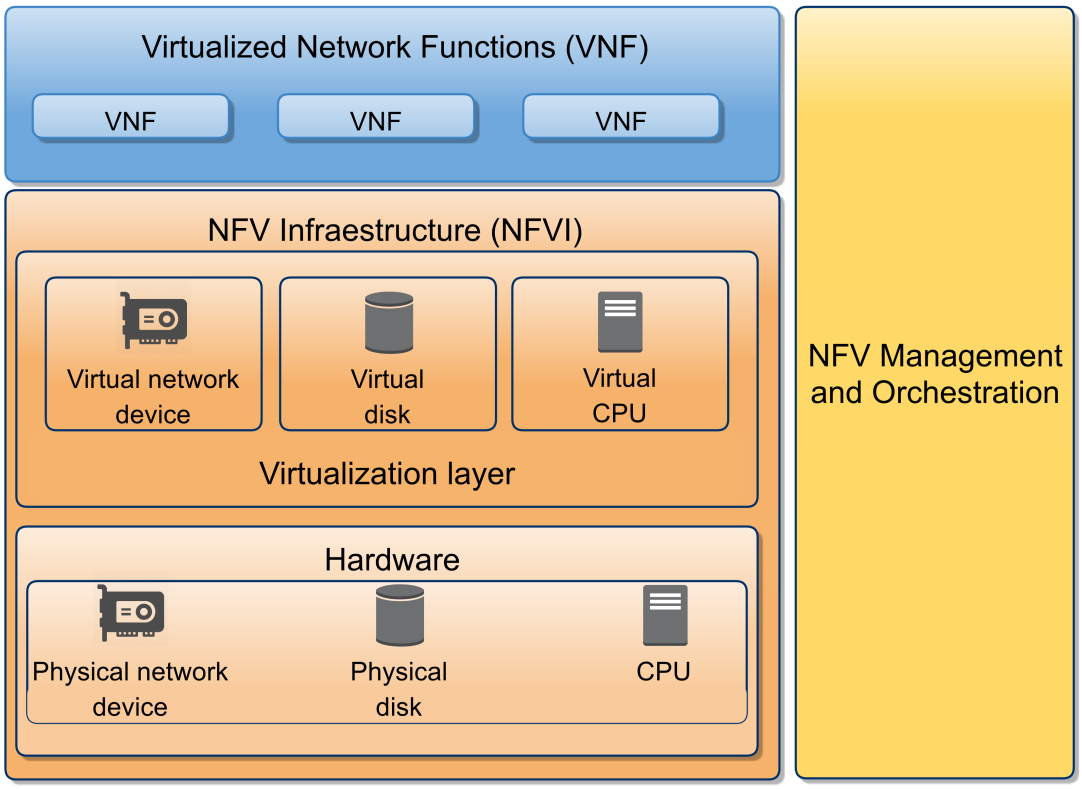


Figura 2.2: Esquema de NFV.

del protocolo *OpenFlow* [McKeown et al., 2008] podría correr como una funcionalidad virtual y comunicarse con un controlador SDN. Por tanto, al igual que ya ocurriera con SDN, NFV presenta unos objetivos de alto nivel bastante similares pero con ciertos matices. Se persigue a grandes rasgos:

(a) Mejorar la eficiencia del capital en comparación con implementaciones hardware dedicadas. Este logro es conseguido mediante el uso de estaciones de propósito general que implementan funciones de red a través de técnicas de virtualización.

(b) Aumentar la flexibilidad al desligar la funcionalidad de la estación física.

(c) Rápida adaptación e innovación al ofrecer servicios basados en software.

(d) Reducir el coste al ser capaces de migrar estaciones de trabajo y apagar completamente estaciones no utilizadas.

Como nota final de NFV, gracias a su uso, la infraestructura es desligada del proceso, favoreciendo modelos de desarrollo independientes, escalables y con unos periodos de puesto en mercado aceptables.

2.2 Aceleración mediante FPGAs

La combinación de PCIe, enlaces de 10 Gigabit Ethernet e interfaces de memoria integradas en plataformas programables son una solución perfecta para obtener un alto rendimiento y baja latencia en el cómputo y tratamiento de la información en redes de telecomunicaciones.

Una gran parte de los algoritmos contienen partes explotables gracias al paralelismo o a la aplicación de *pipeline*. El rendimiento sostenido de una FPGA permite alcanzar resultados óptimos que difícilmente son alcanzables por una plataforma completamente software. La precisión y el control en el manejo de cada camino de datos es total en el hardware donde se controla cómo la información es transmitida en cada ciclo de reloj.

Mientras que arquitecturas CPU *multicore* o las unidades de procesamiento gráfico (GPUs) contienen un número prefijado de bloques lógicos, una FPGA puede ser configurada para ofrecer el ratio justo para el algoritmo particular, garantizando un gran equilibrio entre rendimiento y recursos. Sin embargo, toda esta técnica requiere de un alto grado de conocimiento y unos tiempos de desarrollo generalmente elevados. En conjunción con unas características de almacenamiento más limitadas que una CPU, no es de extrañar que arquitecturas donde la FPGA se aplique como un acelerador sean de gran utilidad. Únicamente una parte de la plataforma es trasladada a la FPGA, mientras que las partes más costosas de implementar, o aquellas donde la aceleración sería más escueta, se conservan en el software. Es el paradigma aplicado por GPUs.

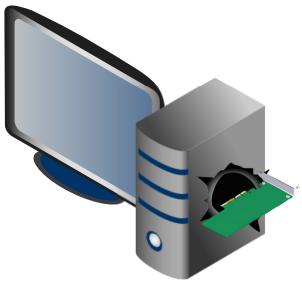


Figura 2.3: Conexión entre PC y FPGA.

En el caso particular de aceleradores basados en FPGA, en comparación con alternativas basadas exclusivamente en CPU, la tasa de transferencia entre la estación anfitriona (PC) y el recurso hardware debe ser considerada. Múltiples son las interfaces actuales para este propósito, tales como UART, eSATA, Ethernet, PCIe o USB. Sin embargo, dadas las altas demandas de las redes multigigabit, la tercera generación de PCIe se impone como medio de transmisión por excelencia en enlaces Ethernet multigigabit, con tasas teóricas de hasta 64 Gbps y soportado por la mayoría de ordenadores modernos. La Figura 2.3 representa la conexión habitual entre ambos dispositivos.

El uso de un acelerador hardware como elemento de extensión de las funcionalidades de un PC, obliga a ampliar a su vez los sistemas operativos actuales con controladores específicos para el dispositivo. Así pues, en la Figura 2.4 se representa el flujo habitual para la preparación del entorno. En primer lugar, a partir de un diseño específico desarrollado para la plataforma FPGA, se genera un *bitstream* (fichero de configuración del hardware reconfigurable) con el que se dota de las competencias oportunas a la tarjeta.

Este *bitstream* debe implementar la lógica necesaria para que el dispositivo sea capaz de interactuar mediante PCIe. Es, en este punto, donde soporte por parte del sistema operativo es requerido. Así pues, tanto controladores como diseños de usuario capaces de interactuar con el sistema son precisos.

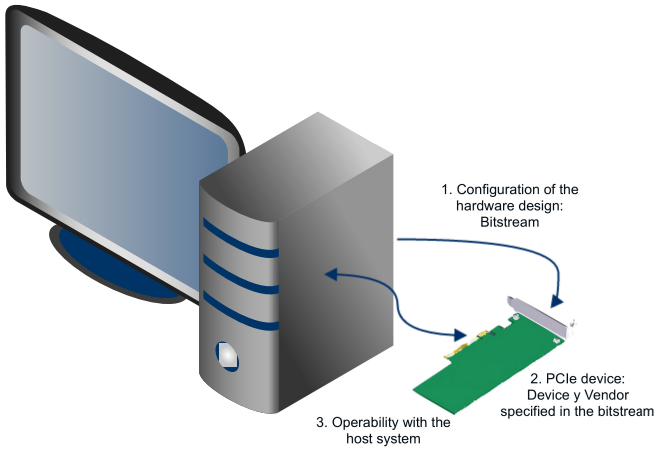


Figura 2.4: Etapas previas a la comunicación PC y FPGA.

En este caso particular, donde un manejo intensivo de los datos es llevado a cabo, competencias para la transmisión, tanto mediante intervención directa de la CPU como por DMA, es esencial. De este modo, tras unas mínimas configuraciones por parte del software, la comunicación de datos es lograda. Los puntos involucrados en un intercambio de información son generalmente:

(a) Los procesos de usuario reservan una región de memoria virtual concedida por el sistema operativo.

(b) El proceso de usuario comunica al controlador del dispositivo FPGA la necesidad de transferir desde (o hacia) esa región información.

(c) El controlador, en modo *kernel*, obtiene las direcciones físicas a partir de las direcciones virtuales y transmite mediante PCIe al diseño hardware la dirección y tamaño a transferir.

(d) En el diseño hardware, se interpretan estas solicitudes. Se configura el motor de DMA y comienza la operación.

(e) Al terminar el proceso, se genera una interrupción para avisar de la finalización de la operación.

(f) El diseño a nivel de driver comunica la terminación al proceso de usuario (por el método seleccionado por el desarrollador: operación bloqueante, consulta periódico del estado del driver, etc).

(g) El proceso de usuario en software dispone de los datos sobre los que seguir procesando o, si la aplicación requirió que los datos fueran enviados a la FPGA, ya se conoce que la operación ha culminado.

Este enfoque ha sido clásico, ya no únicamente a nivel de FPGAs, y ha sido explotado en otros entornos de computación paralela como en la plataforma Xeon Phi de Intel o las ya mencionadas unidades de procesamiento gráfico. Sin embargo, cierta dependencia de la máquina anfitriona es necesaria, en tanto que el sistema operativo debe conocer y soportar la existencia del acelerador. Los recursos de los que se hace uso son los originarios de la máquina física, por lo que es un escenario lejano al deseable en la actualidad, ni tan robusto ni dinámico como las nuevas exigencias demandan.

2.3 Plataforma virtual alternativa

En este apartado se plantea una arquitectura basada en FPGAs capaz de mutar y adaptarse a las fluctuantes necesidades del mercado. Con una total abstracción de la plataforma subyacente, los ciclos de desarrollo del software y el hardware quedan claramente diferenciados. Una arquitectura inspirada en la especificación de NFV se contempla y propone como solución para acortar los periodos de desarrollo.

En el escenario inicial, con un acelerador hardware basado en FPGA, el sistema operativo posee acceso directo a los recursos del equipo. Sin embargo, como se muestra en el Apéndice A.1, la virtualización y emulación de recursos es un proceso que lleva estudiándose durante varias décadas. En esta situación, el uso de máquinas virtuales, abstrae al sistema operativo invitado de los recursos reales.

Una primera aproximación pasa por dotar al sistema operativo anfitrión de las capacidades para la configuración de la plataforma hardware. Una vez que las conexiones físicas se encuentran habilitadas, los distintos entornos virtuales pueden interactuar con el acelerador. Esta arquitectura aparece representada en la Figura 2.5. El sistema virtualizado, ya no depende

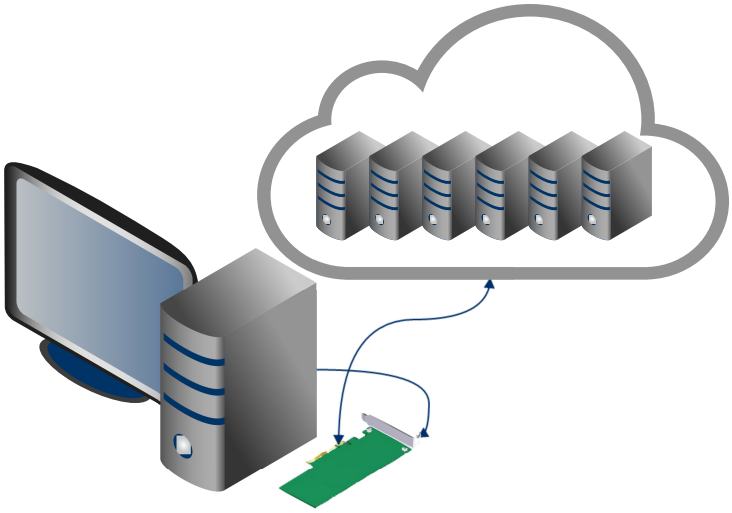


Figura 2.5: Entorno virtualizado basado en acelerador FPGA (I). Uso de máquinas virtuales.

nunca más del almacenamiento físico o CPU, en tanto que el monitor de máquinas virtuales provee de esta capa de independencia. Sin embargo, el recurso programable sigue siendo único y su compartición entre distintas VMs no es posible. En los Apéndices A.2 y A.3 se detallan las tecnologías *PCI passthrough* y SR-IOV que permiten a una función física figurar como varias funciones virtuales. Adicionalmente, *PCI passthrough* colabora a disminuir la penalización por virtualización al asegurar que una función virtual es conectada directamente a una VM con una menor intervención por parte del hipervisor.

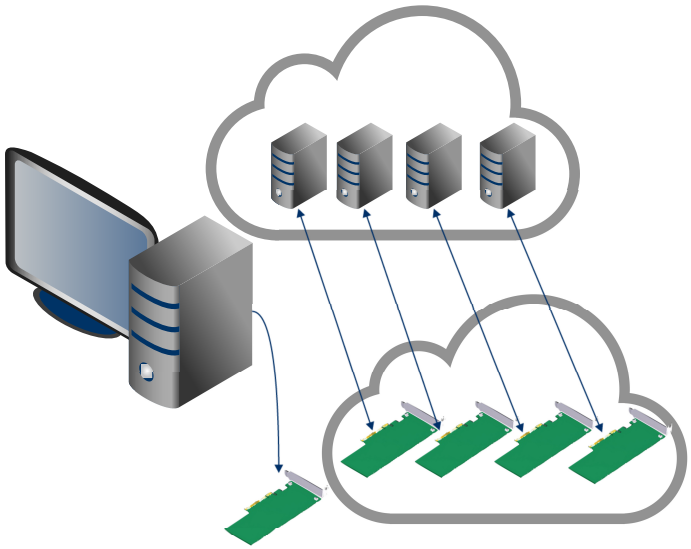


Figura 2.6: Entorno virtualizado basado en acelerador FPGA (II). Uso de máquinas virtuales y SR-IOV.

Incorporando estas técnicas al modelo inicial, tal y como se muestra en la Figura 2.6, tanto recursos de almacenamiento, memoria y CPU son aislados de los utilizados por el sistema base en una estación virtualizada. La compartición del dispositivo, un campo menos explotado, es obtenida a partir de su implementación con los propios recursos lógicos y la tecnología SR-IOV. Así pues, en un momento inicial el ordenador anfitrión configura la FPGA para que habilite el soporte de virtualización. Una vez que esta tarea ha sido lograda, cada VM reconoce su propio dispositivo figurado (ya que no es físico) y puede interactuar de manera independiente con él.

En este momento, realizando la analogía con NFV (Figura 2.2), los elementos quedan determinados por:

- La infraestructura NFV queda integrada por el hipervisor de máquinas virtuales, el equipo físico, la tarjeta reprogramable y el soporte por parte de ésta para SR-IOV.
- La plataforma capaz de ejecutar funciones virtuales se corresponde con un sistema operativo corriendo de manera invitada sobre la plataforma original y con un módulo controlador capaz de manejar el dispositivo abstracto conectado en exclusiva a este entorno (controlador virtual).
- El árbitro de NFV es el controlador de la máquina anfitriona, capaz de habilitar o deshabilitar unidades virtuales bajo demanda. A su vez, el hipervisor juega un papel fundamental, ya que la suspensión de la plataforma en su totalidad es una tarea que recae sobre él. La sincronización entre los procesos que acceden a un recurso virtual de la plataforma FPGA se realiza en la propia placa o, alternativamente, en el módulo físico a necesidad del problema.

Entidad	Equivalencia
Funcionalidades NFV	VM ejecutada sobre la arquitectura anfitriona + Controlador virtual FPGA + Software NFV
Infraestructura NFV	Equipo físico con sistema operativo capaz de aplicar virtualización + FPGA configurada con soporte para SR-IOV
Árbitro	Diseño FPGA + Controlador físico FPGA + VMM

Tabla 2.1: Equivalencia NFV y arquitectura propuesta.

De los anteriores conceptos, merece la pena detenerse en la figura del árbitro. Supóngase un acelerador para problemas de red. Imagínese que el problema a tratar es *deep packet inspection* para el reconocimiento del protocolo intrínseco a una comunicación. Esta tarea será realizada por las máquinas virtuales sostenidas sobre la arquitectura anfitriona. La FPGA cuenta con un total de 2 interfaces de red y existen 4 VMs y 4 funciones virtuales. Inicialmente, todas las VMs requerirán de datos para trabajar y aquí es donde el árbitro toma lugar. Existen distintas políticas de planificación. Por ejemplo, la VM1 y la VM2 trabajan cada una sobre la interfaz Ethernet correspondiente. Como sólo hay dos hábiles, el árbitro opta por suspender las otras dos estaciones infrautilizadas.

Esta aproximación puede ser loable si la carga del sistema es elevada o se prima el ahorro energético aunque pudiera no ser siempre la situación. Supóngase el caso donde interesa favorecer el alto rendimiento. En tal caso, la información recibida por una interfaz podría ser puesta a disposición de la máquina virtual que se localizase ociosa en ese momento, aplicando una política de balanceamiento de carga que asegure el trabajo equilibrado entre todas las estaciones. Esta distribución se puede realizar tanto a nivel de módulo controlador físico o a nivel de diseño FPGA. El segundo escenario es más eficiente, aunque conlleva unos ciclos de desarrollo más largos. Por tanto, esta cuestión de diseño queda reservada a las necesidades concretas de la plataforma elaborada.

Una última mejora a la plataforma, es la incorporación de reconfiguración parcial del dispositivo. En esta versión, el sistema anfitrión habilita el soporte para SR-IOV en la FPGA. Seguidamente cada VM configura un diseño propio en la tarjeta, de modo que cumplirá una funcionalidad particular. Independencia tanto del diseño original como de la funcionalidad que el acelerador desempeña se asegura, en el sentido de que para cada estación virtual una tarea distinta se está desempeñando (ver Figura 2.7).

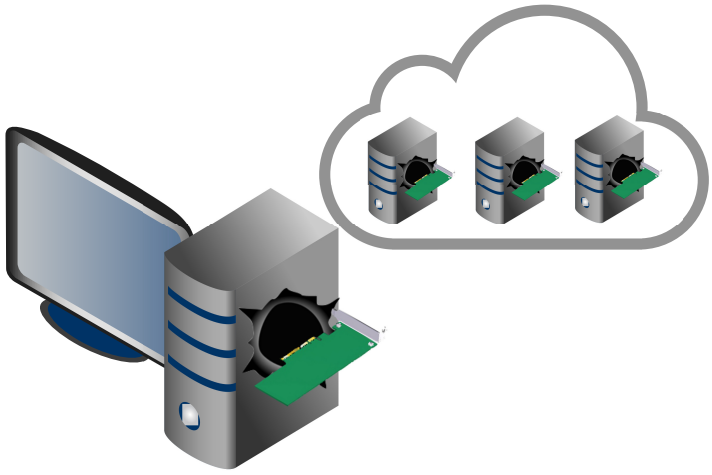


Figura 2.7: Entorno virtualizado basado en acelerador FPGA (III). Uso de VMs, SR-IOV y reconfiguración parcial.

Este último escenario permitiría completa independencia de la estación anfitriona y además permitiría explotar los recursos de la FPGA (abstraídos del modelo concreto) de una manera satisfactoria.

2.3.1 Ejemplo de referencia

En esta subsección se va a identificar un problema real, identificando y contextualizando los beneficios de la arquitectura. Supóngase que la tarjeta reprogramable usada contiene un

total de 4 interfaces de red. El mismo número de estaciones virtuales se crean sobre el sistema anfitrión y se configuran mediante un *bitstream* el soporte para SR-IOV y 4 funciones virtuales.

Inicialmente la función física (que es replicada como funciones virtuales) cumple con el funcionamiento de una NIC convencional. Es decir, transfiere los datos hacia la máquina virtual asociada o devuelve los datos a la red si la VM así lo indica.

Todas las funciones virtuales son análogas. Cumplen la misma funcionalidad y cada una está ligada a una interfaz física. El árbitro no se ve obligado a suspender ninguna VM y la asociación entre máquina virtual e interfaz física se realiza en el diseño FPGA.

Supóngase que se precisa realizar un cortafuegos pero que las reglas varían en cada una de las interfaces. Los programas corriendo a nivel de usuario en una VM únicamente se encargan de aplicar la búsqueda de patrones correspondiente a las listas de bloqueo. Cada VM contiene una lista diferente pero el código es equivalente en todas las estaciones. Cambiar una regla consistiría en actualizar un equipo virtual particular sin que ello afectase al resto de elementos. Éste es el escenario de la Figura 2.6.

Sin embargo, por un aumento inesperado del tráfico en la última semana, el sistema está a pleno funcionamiento y parece ser imposible analizar todo el contenido. Como el desarrollo se ejecuta a nivel software es muy sencillo aplicar un muestreo estadístico de los paquetes que permita salir de la situación en un tiempo de actuación ínfimo. Como arquitectura flexible la aportación ideal sería que el acelerador hardware permitiera la aplicación de reglas directamente. Sin embargo, la problemática radica en que cada estación tiene necesidades distintas.

La reconfiguración parcial ofrece, en este punto, que cada VM implemente usando los recursos lógicos de la tarjeta sus propias reglas, de manera invisible a las demás y solventando los problemas de rendimiento. No obstante, dado que la implementación de todas las firmas puede ser una tarea muy laboriosa, la descarga de un número más pequeño (las más costosas en software, las más comunes, etc.) es factible y totalmente independiente de la estación anfitriona. Aplicando esta técnica un total de 5 *bitstreams* son precisos en total. El *bitstream* inicial que habilite la configuración del soporte para la virtualización y otro adicional por cada VM para configurar las reglas concretas a implementar.

2.3.2 Diseño de referencia

Motivada y justificada la arquitectura propuesta, a lo largo del documento se va a contemplar un escenario real, donde todos los puntos abarcados hasta el momento van a ser sometidos a evaluación y medida de su viabilidad para una aplicación real más allá de un plano teórico. El objeto es la creación y virtualización de una NIC mediante hardware reconfigurable, sin soporte para reconfiguración parcial pero con el resto de características plenamente operativas.

3 Estado de la técnica

En este mismo instante un gran interés por las tecnologías NFV se ha levantado. Las ventajas de desplazarse desde soluciones hardware propietarias a estaciones virtualizadas en software son múltiples y bien conocidas:

(a) Reducción de costes y consumo energético.

(b) Tiempos de puesta en mercado y despliegue de la aplicación en periodos más breves.

(c) Flexibilidad de una plataforma para su uso por diferentes usuarios y/o servicios.

(d) Complemento a las SDN. Es decir, tanto el control de la transmisión como el procesamiento puede ser tratado en la CPU.

Con la inclusión de FPGAs en este proceso, la funcionalidad virtual se compone tanto del software a nivel de CPU como del diseño en lenguaje de descripción hardware (HDL). Cuando la información transferida por la red es recopilada por la unidad de coprocesamiento, estos datos pueden ser tratados exclusivamente por el hardware reconfigurable o, por el contrario, ser transferidos a la estación software.

El sistema admite dinamicidad en el modelo de computación, permitiendo que las aplicaciones sean sostenidas exclusivamente en hardware, completamente en la CPU o en una mezcla de ambas. En esta característica radica una de las principales ventajas al comparar el modelo con las soluciones virtualizadas sobre NICs convencionales, donde la plataforma hardware no admite soporte para el despliegue de nuevas funcionalidades no previstas inicialmente. Para solventar esta rigidez, tradicionalmente se han tratado de explotar todos los recursos disponibles a nivel de CPU para que la extensión se aplicara a nivel de software. Las aplicaciones implementan paralelismo, procesamiento en lotes de los paquetes, suprimen los mecanismos de intercambio de mensajes (se eliminan técnicas como la generación de interrupciones para aplicar una espera activa que reduzca al máximo los tiempos de inactividad en detrimento de eficiencia en términos energéticos y de cómputo (Intel DPDK [Dumitrescu, 2008])) y aprovechan de manera óptima la memoria (modelos *zero-copy*).

Adicionalmente, es común paliar los retrasos asociados al sistema operativo evitando hacer uso de aspectos tan genéricos como la pila de red, como se demuestra en [Rizzo, 2012]. Por otra parte, en el momento en el que un entorno virtualizado es la plataforma objetivo, el número de intercambios de contexto entre máquina anfitriona e invitada deben ser reducidos si el rendimiento es una prioridad. Una amplia lista de claves se provee en [Rizzo et al., 2013] donde

se explican métodos para acelerar el proceso en diversos entornos: donde modificaciones sobre la VM pueden ser aplicadas y aquellos donde también existe libertad para configurar el hipervisor.

En esta tendencia de optimización de grano fino en la virtualización, y con objeto de mejorar la posibilidad de migración de los servicios a estaciones menos saturadas, [Garzarella et al., 2015] propone el uso conjunto de SR-IOV con modificaciones al hipervisor en tarjetas de red. De este modo, un dispositivo real crearía múltiples funciones virtuales. Éstas son moderadas por el controlador instanciado en el sistema anfitrión (*driver físico*), mientras que las estaciones virtuales interaccionan con este elemento para obtener acceso a los recursos. Aunque esta mejora en cuestión de independencia de la plataforma es loable, existe el inconveniente de que las comunicaciones entre ambos sistemas no juegan a favor de la eficiencia. Hay que asegurarse de que ninguna notificación es perdida por cuestión de carreras, la latencia que incorpora y, en el peor de los casos, la posibilidad de que un *thread* dormido sea planificado en otro procesador. En tal caso el hilo partirá de una memoria caché con datos no reutilizables y, de manera genérica, el retrado será del orden de microsegundos.

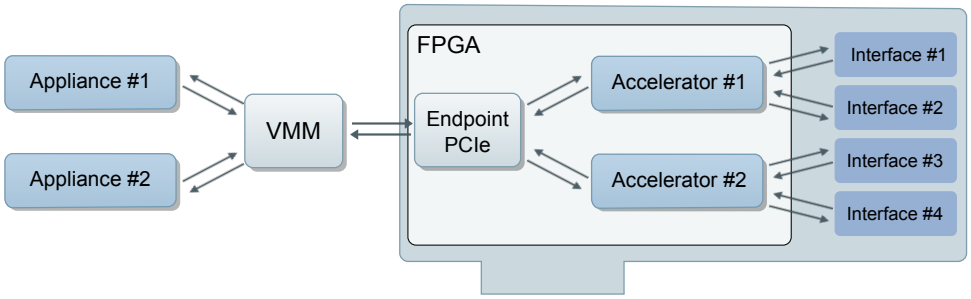


Figura 3.1: Ejemplo de desarrollo de un acelerador de red en FPGA: conexiones físicas

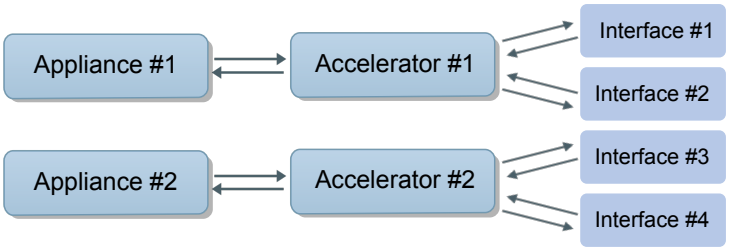


Figura 3.2: Ejemplo de desarrollo de un acelerador de red en FPGA: conexiones lógicas

Desafortunadamente, aunque estas aproximaciones han funcionado relativamente bien para enlaces de hasta 10 Gbps, con el aumento de los enlaces en el troncal de internet se espera que esta moda no disfrute de la misma validez. El *passthrough* de un dispositivo directamente a una VM favorecerá que la máquina virtual pueda explotar los recursos casi de manera nativa ya que la conversión entre direcciones virtuales y físicas es gestionada mediante la IOMMU [Ben-yehuda et al., 2006] sin intervención de la estación anfitriona. Así pues, aceleradores sobre las

interfaces de red en conjunción con los mecanismos de virtualización explicados en la Sección 2 aseguran una opción viable aunque con carencias actuales para la migración en directo. La comunicación entre componente y máquina virtual se realiza mediante PCIe sin intervención el *host* y una visión tanto lógica como física es representada en las Figuras 3.1 y 3.2 respectivamente.

Aplicación	**Referencia**
Cortafuegos	[Wicaksana y Sasongko, 2011]
Sistema de detección de intrusiones	[Bu y Chandy, 2004] [Sourdis et al., 2007] [Mitra et al., 2007]
Switching basado en contenido de paquetes	[Kachris y Vassiliadis, 2006]
Extracción de campos de paquetes	[Attig y Brebner, 2011]
Selección de paquetes basados en *IP*	[Jiang y Prasanna, 2013] [Ganegedara y Prasanna, 2013]
Inspección del contenido de paquete (*deep packet inspection*)	[Yang y Prasanna, 2013] [Dharmapurikar et al., 2003]
Sistema antivirus (gusanos)	[Lockwood et al., 2004]

Tabla 3.1: Aplicaciones de red susceptibles de ser implementadas en FPGAs.

La lista de aceleradores es amplia y algunos ejemplos quedan descritos en la Tabla 3.1. En todos los casos, el mayor reto, dejando de lado la selección del modelo de programación, es el intercambio efectivo de información entre máquinas virtuales y aceleradores hardware. Como ya fuera mencionado, dos son las claves para conseguir reducir el *overhead* en las comunicaciones: SR-IOV y *PCI passthrough*, técnicas definidas en la especificación de PCIe [SIG, 2014]. Mientras que la primera habilita la creación de múltiples funciones virtuales de un mismo acelerador, el segundo asegura la asociación dispositivo-máquina.

3.1 Trabajo relacionado

Si se deseara localizar un entorno de desarrollo que cumpliera con las especificaciones íntegras de virtualización de aplicaciones de red para su manejo en software, la oferta está muy limitada y son soluciones muy rígidas. No existe una plataforma abierta ni un estándar que se haya impuesto sobre las demás alternativas. Las ofertas sustentadas en hardware reconfigurable tampoco han sido la principal apuesta hasta el momento por la carencia de desarrollos previos que agilicen el desarrollo. Así pues, distintos fabricantes cuentan con iniciativas propietarias que, aunque válidas para escenarios puntuales, no siempre son la mejor opción en cuanto a tratamiento de tráfico se refiere. Entre las alternativas más orientadas al mundo em-

presarial merece la pena destacar:

(a) La apuesta de *Xilinx* en este proceso de abstracción del hardware es SDNet [L. Wirbel, 2014]. SDNet permite la creación de sistemas de procesamiento de paquetes mediante la compilación de código de alto nivel orientado al paralelismo, OpenCL.

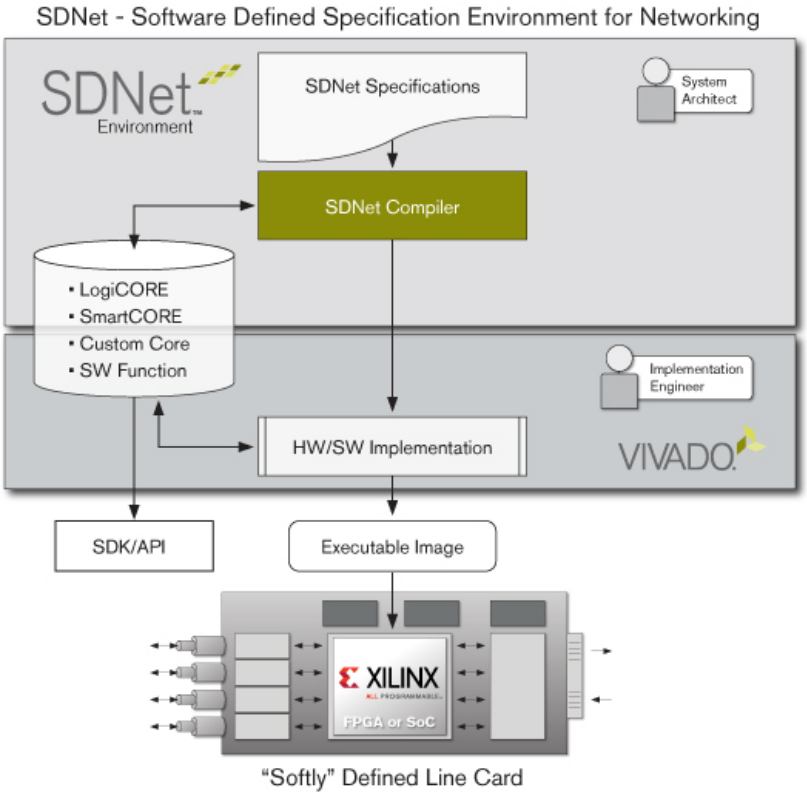


Figura 3.3: Entorno de desarrollo de aplicaciones de red con la herramienta SDNet de Xilinx.

Aunque los costes de producción pueden ser recortados drásticamente al ser capaces de evitar el uso de lenguajes HDL sigue existiendo el problema de la comunicación con la FPGA. Considérese de nuevo el ejemplo de un *firewall* que es implementado con SDNet. La aplicación debe aceptar la creación/supresión/modificación de las reglas en tiempo de ejecución y que una nueva imagen sea generada para adaptar el diseño en tiempo real es, con la versión actual de la herramienta, inviable.

(b) *EZchip Technologies*, proveedores de motores de alto rendimiento para la redirección de paquetes, han abandonado el enfoque clásico de un hardware dedicado cumpliendo la totalidad de la funcionalidad de manera aislada. De manera sustitutiva, el procesamiento multihilo bajo sistemas Linux parece ser la apuesta [EZChip, 2014].

La tarjeta de desarrollo *Tilera*, de la misma compañía, ofrece al usuario un total de 64 *cores* para la resolución del problema que se contemple. Su programación se realiza a nivel software y un procesamiento paralelo y masivo favorece un gran rendimiento por parte

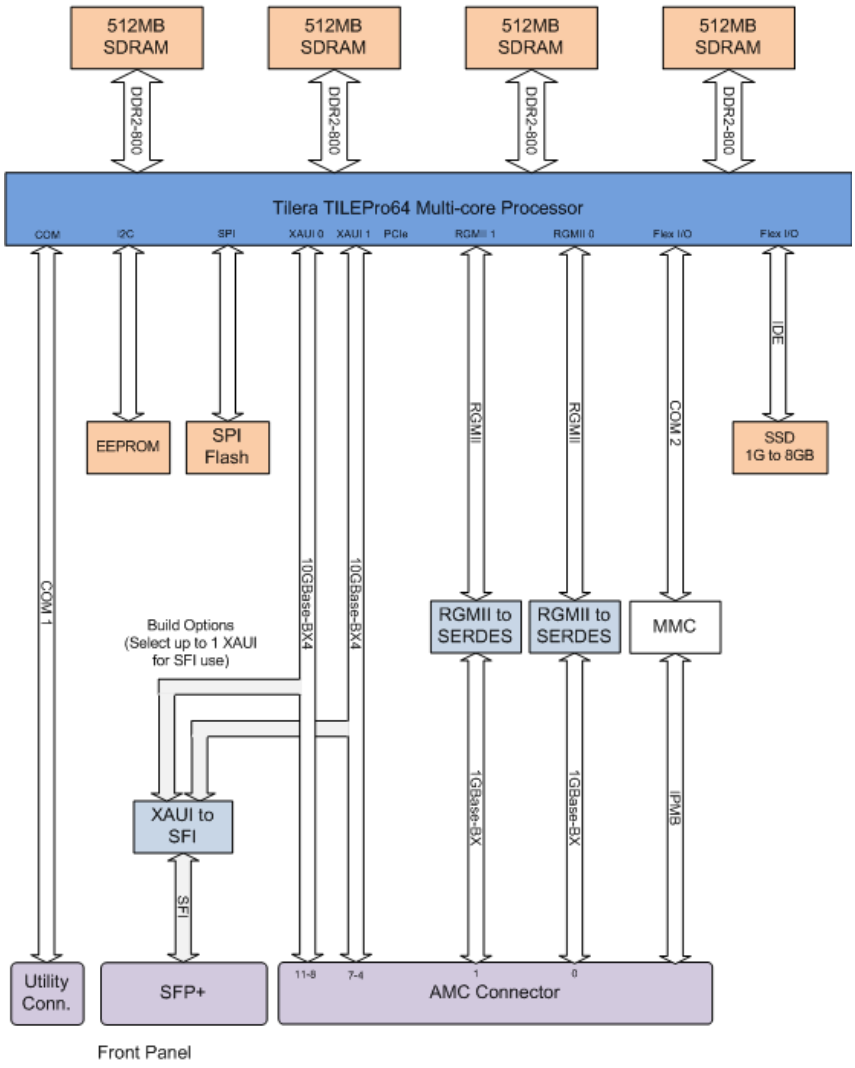


Figura 3.4: Diagrama de bloques para la tarjeta *Tilera* TPM-100-BD.

de la arquitectura. No obstante, no debe olvidarse que las posibilidades de aceleración hardware están muy limitadas y la mayoría del cómputo es realizado exclusivamente por la CPU. Aunque se reducen los tiempos de puesta en el mercado, la flexibilidad es restringida y la solución ligada a un vendedor particular (pérdida de independencia e imposibilidad de utilizar *commodity hardware*). En la Figura 3.4 se muestra el diagrama de bloques de la solución de *EZchip Technologies*.

(c) *Marvell*, con la adquisición *Xelerated*, o *Cavium* con su familia de procesadores *Octeon*, siguen la misma premisa que *EZchip Technologies*, ofreciendo las capacidades más demandadas en el tratamiento de tráfico de red (módulos de encriptación, rutinas para inspeccionar el contenido del paquete en busca de firmas,etc.) extensibles mediante soft-

ware creado para la arquitectura seleccionada. Por tanto, al igual que pasara con la tarjeta *Tilera*, la independencia del procesamiento y la arquitectura es prácticamente nula.

Dejando de lado las alternativas comerciales, en el ámbito más investigador si que aparece alguna solución sustentada en FPGAs con requisitos parecidos. Por ejemplo, *Elastic AH* [Nobach y Hausheer, 2015] construye una aproximación compuesta de un *pool* de funciones junto con un número predefinido de módulos aceleradores en hardware. Un programa software que integra esta arquitectura está integrado, por tanto, de la NFV asociada y de las aplicaciones encargadas para la comunicación con los módulos aceleradores hardware. No obstante, existe una limitación en el uso del hardware reconfigurable y es la exclusividad de éste para la descarga de procesamiento. Esta infrautilización del recurso penaliza negativamente en los costes de la solución dado que su emplazamiento como tarjeta de red es completamente viable [Zazo et al., 2014].

Por otra parte, plataformas íntegras en FPGAs para NFV también han sido exploradas [Kachris et al., 2014]. Aunque en esta ocasión se cuenta con soporte para tratar con SDN, la plenitud de la lógica de funciones virtuales y manejo de datos están controlados por la plataforma hardware. La posibilidad de extender la computación sobre los datos con programas software no es viable y sería una característica completamente recomendable.

4 Diseño e implementación

Tras un análisis de las tecnologías actuales que ofrecen soporte a la virtualización, destacando el novedoso concepto de SR-IOV, es objeto de este trabajo elaborar una propuesta de arquitectura para la virtualización de tarjetas de red de altas prestaciones.

La actual sección presenta un enfoque ascendente de la arquitectura, partiendo desde el hardware reconfigurable para ir tratando de manera incremental los diseños hasta culminar con aquel que permita una virtualización del dispositivo mediante SR-IOV. La configuración software y controladores de dispositivos necesarios para cumplir la funcionalidad requerida son detallados a continuación.

4.1 Plataforma de desarrollo

Debido al alto coste temporal y monetario que el desarrollo de una tarjeta dedicada supondría, su implementación es adaptada a FPGAs. No debe olvidarse la existencia de proyectos completamente orientados al procesamiento del tráfico de red como NetFPGA [Gibb et al., 2008], perfectamente viables para desempeñar esta funcionalidad, pero con un principal inconveniente: el encarecimiento de su precio en su implantación en entornos industriales.

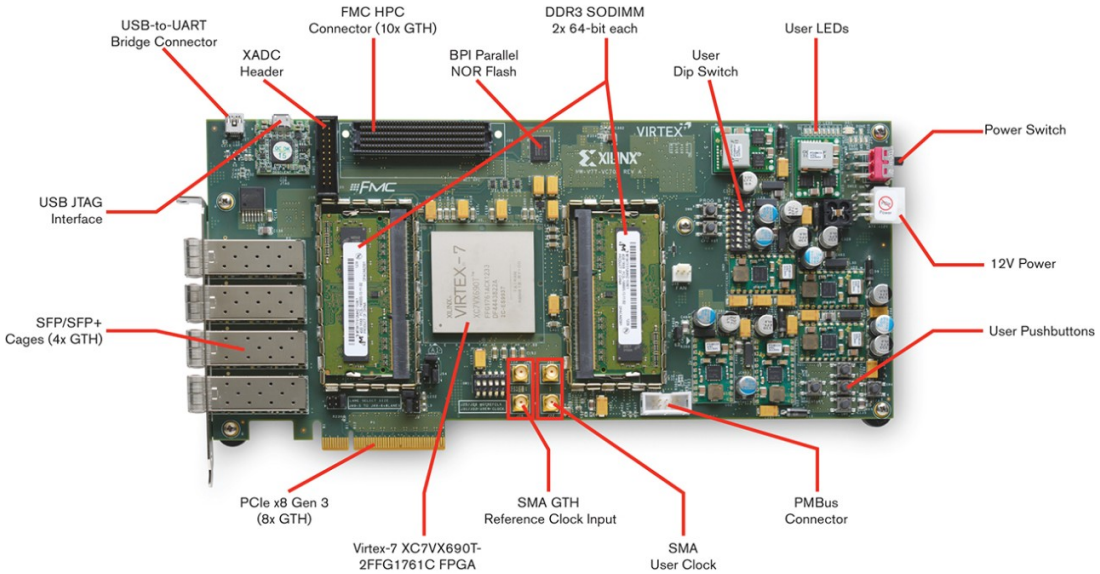


Figura 4.1: Placa de desarrollo VC709 de Xilinx.

Como sustento para todo el proceso de virtualización, la plataforma hardware reconfigurable seleccionada para la aplicación es la placa VC709 (ver Figura 4.1), capaz de lograr un alto rendimiento gracias a la FPGA Virtex 7 VX690T de Xilinx.

Junto al coste moderado de la plataforma en ámbitos empresariales, la segunda motivación para su elección es la inclusión de 8 líneas PCIe de generación 3.0. Un total de 4 módulos SFP+ (4 transceptores 10 GbE) y 8 GB DDR3 SODIMM aseguran los mecanismos necesarios para poder trabajar a tasa de línea en enlaces multigigabit. Adicionalmente, existe el soporte para el *core* DMA desarrollado por Northwest Logic. Sin embargo, por limitaciones en aspectos de virtualización, ha sido reemplazado por un desarrollo propio en la versión final del diseño.

4.1.1 Implementación de una tarjeta de red con hardware reconfigurable

Como ya se mostrara en [Zazo et al., 2014], la viabilidad de la reproducción/captura de tráfico por parte de un dispositivo hardware reprogramable es una tarea viable y bastante ágil dado el modelo de programación HDL basado en reutilización de bloques lógicos de datos (IP cores). Es decir, la creación de una tarjeta de red basada en FPGAs, aunque compleja dado que involucra trabajar a un bajo nivel, es una tarea resuelta con anterioridad a este documento y no será el principal objeto de exploración en este trabajo. Sin embargo, es conveniente recalcar los aspectos más notorios involucrados en este tipo de aplicaciones.

En particular, para la plataforma de desarrollo citada, el manejo de las interfaces de red hasta una tasa de 10 Gbps es asequible gracias al uso de los IP cores *10 Gigabit Ethernet PCS/PMA (10GBASE-R)* y *10 Gigabit Ethernet Media Access Controller* de Xilinx. El primero presenta la lógica necesaria para la abstracción del medio físico; el segundo provee la capa del nivel de enlace. Si niveles superiores del modelo fueran requeridos, tradicionalmente ha sido el propio sistema operativo el encargado de realizar estas operaciones, por lo que un *driver* convencional de red puede ser adaptado para tal propósito.

No obstante, en entornos de alto rendimiento niveles superiores en el modelo *Open Systems Interconnection model (OSI)* son generalmente despreciados [Moreno et al., 2014]. Esta decisión de diseño se sustenta en el alto coste requerido por los protocolos orientados a sesión (como el protocolo de control de transmisión (TCP)). La reconstrucción de fragmentos, que implica albergar la información parcial en memoria, es una tarea de difícil acceso a nivel hardware. Motivándose en que no todas las soluciones software requieren de este procesamiento se carece, por tanto, de su desarrollo dejando tal libertad a niveles superiores del sistema (software).

A su vez, otra pieza clave en todo el proceso es la comunicación de los datos mediante PCIe al sistema anfitrión. Esto se consigue gracias al IP core *7 Series FPGAs Integrated Block for PCI Express* [Xilinx, 2015] que cumple con la funcionalidad de *endpoint*, permitiendo al dispositivo

figurar como un elemento PCIe. El manejo del espacio de configuración, así como la tarea de anunciar las competencias, queda resuelto por este IP core. Por el contrario, la encapsulación de la información para su transmisión por el enlace multigigabit no está implementada. El *core* de DMA de Northwest Logic abstrae del nivel de Transaction Layer Packet (TLP) y facilita tanto la transmisión directa de datos (CPU encargada de la lectura/escritura de los datos) como las operaciones de DMA, con la consecuente liberación de carga al sistema global. Con estos IP cores una tarjeta de red se consigue implementar con un diseño como el de la Figura 4.2. Las características más destacables del modelo representado por diagrama de bloques son:

- Se han contemplado un total de 4 enlaces diferentes de red, ajustable al modelo del kit VC709 y los 4 transceptores SFP+.
- Por cada enlace existe un par diferencial para la transmisión de datos y otro segundo destinado a la recepción.
- La configuración de los dispositivos externos se realiza mediante un bus serie de baja frecuencia. Este proceso puede variar completamente dependiendo de la tarjeta objetivo. Si el bus seleccionado es *Management Data Input/Output (MDIO)*, es el propio controlador *media access control (MAC)* el que provee de dicha funcionalidad. Aprovechando la interfaz AXI4-Lite del IP core MAC, los datos transmitidos mediante ésta son procesados y convertidos, si procede, de acuerdo al estándar MDIO. De manera análoga existen cores reutilizables que simplificarían la adaptación a otros entornos como aquel donde IIC es el bus empleado para la configuración de periféricos.
- El proyecto de Northwest Logic simplifica en gran medida cualquier comunicación mediante PCIe, poniendo a disposición del usuario un total de 3 interfaces:
  - Interfaz de configuración (AXI4L-MGMT). Permite que mediante escrituras de la CPU en regiones bien conocidas se pueda cambiar el comportamiento del sistema. Ejemplos concretos son la planificación de balanceo de carga, el tamaño máximo de MTU del sistema, direcciones MAC, etc.
  - Interfaz de recepción de datos(AXIS_C2S). Los octetos volcados a través de esta interfaz serán puestos a disposición del diseño software mediante transferencias DMA.
  - Interfaz de transmisión de datos(AXIS_S2C). La información recibida a través de esta interfaz será la transmitida del diseño software mediante transferencias DMA.
- En última instancia, los datos de cada una de las interfaces de red pueden ser accedidos a través de un protocolo genérico: AXI4-Stream. Así pues, es tarea del diseño anotado como *user app* procesar la información y ponerla a disposición del *core* DMA. La manera en que esto se logra en recepción (información volcada sobre la interfaz AXIS_C2S del *core* DMA) es mediante la utilización de tantas colas *first in first out (FIFO)* como enlaces se estén monitorizando.

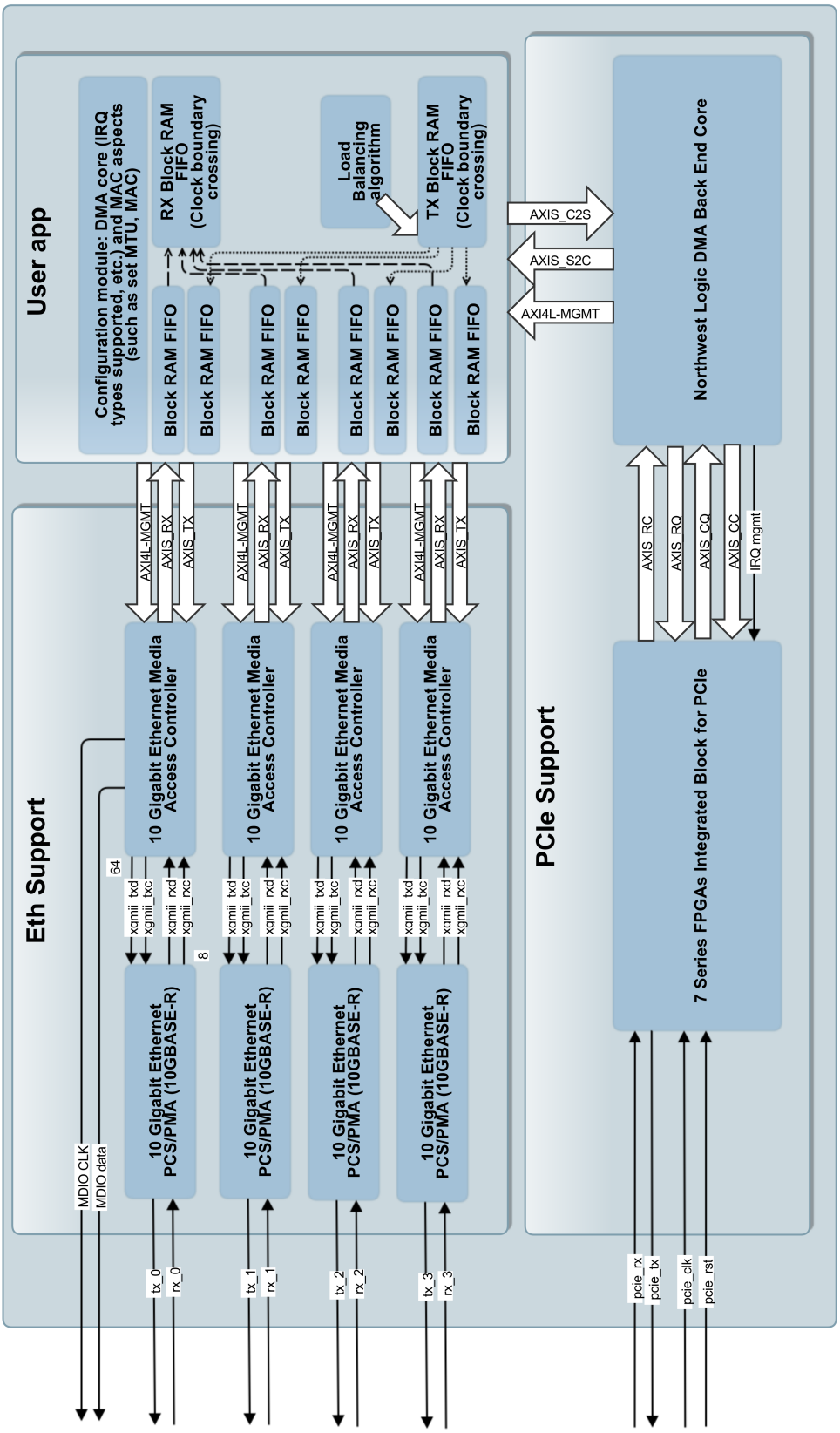


Figura 4.2: Diagrama de bloques de una tarjeta de red implementada bajo hardware reconfigurable de Xilinx.

Posteriormente, cada una de estas memorias es consultada, de modo que si existen datos, son exportados a otra segunda FIFO encarga de realizar el cambio de dominio de reloj. Este cambio de dominio es necesario dadas las diferentes exigencias de los enlaces multigigabit de red (156.25 MHz, ancho bus de datos 64 bits) y las necesidades a nivel de PCIe (250.0 MHz, ancho bus de datos 256 bits). En transmisión (información proveniente de la interfaz AXIS_S2C del *core* DMA), toda la información es guardada en una primera FIFO. Posteriormente, de acuerdo a la política de planificación seleccionada se transmitirán los paquetes por la interfaz que corresponda. En cualquier caso, la implementación variará de acuerdo a las necesidades. De modo que un usuario particular podría tener la necesidad de reproducir todo el tráfico por todas las interfaces o, por el contrario, únicamente por aquella menos saturada. Estas decisiones, perfectamente configurables por parte del *driver*, quedan contempladas en el diagrama bajo el algoritmo de balanceo de carga.

4.1.2 Generación de un módulo DMA alternativo a las opciones comerciales

Como se ha dejado entrever, el *core* DMA de Northwest Logic incluido junto al kit de conectividad como *hard core* no es plenamente funcional para fines de virtualización. En otras palabras, se limita en gran medida la modificación de su funcionalidad debido a las licencias asociadas al desarrollo. Para el caso concreto de SR-IOV es imprescindible tener la capacidad de distinguir entre el origen de las fuentes (funciones virtuales (VFs)) de modo que se entreguen los datos de manera correcta. Sin embargo, esta tarea es completamente opaca al diseño final del usuario.

Para ilustrar la problemática, supóngase que se desea transferir desde la memoria del host n regiones asociadas a distintas máquinas virtuales cumpliendo que el tamaño de cada región es mayor al tamaño del *read request*. Es decir, cada región se va a solicitar en fragmentos de tamaño *read request* y éstos pueden ser mezclados en su entrega por el sistema (o, directamente, desde su solicitud por parte de la FPGA). Nace la problemática natural de separar cada pieza de datos recibida con la aplicación hardware de usuario a la que debe ser enviada.

Aunque soluciones de más alto nivel puedan ser diseñadas, como la idea de añadir una cabecera a los datos, su implantación se vuelve dependiente de la arquitectura al necesitar del valor del *read request*. Una cabecera por cada bloque de datos tendría que ser anexada y dicha acción debería ser realizada por el programa software que genera los datos. Esto limita en gran medida la escalabilidad de la plataforma y, desde un punto de vista lógico, se entremezclan conceptos funcionales diferenciados (generación de datos con el medio de transmisión).

Como prueba de concepto su validez quizá estuviera justificada. Sin embargo, en este trabajo se ha decidido ir un paso más adelante desarrollando un IP core con dicha funcionalidad. Un proceso que a priori puede pasar desapercibido, la encapsulación de la información en TLPs

y su tratamiento, no es una tarea trivial. Con el objetivo principal de obtener el máximo rendimiento, una versión operable en PCIe 3.0 y con hasta un total de 8 líneas es detallada en el siguiente punto.

Comunicaciones mediante PCIe

Aunque el IP core *Xilinx 7 Series Integrated Block for PCI Express* evita el proceso de reconstruir un paquete de las líneas físicas de PCIe, identificar y generar paquetes en un formato propio (con características comunes a un TLPs) es una tarea que debe ser llevada a cabo por un agente ajeno. En este punto de identificar y generar paquetes es donde se permite aplicar ciertas reducciones sobre la especificación original de PCIe [SIG, 2014].

Partiendo de la premisa de que el entorno únicamente requiere transferencia entre FPGA-CPU. De este modo, la comunicación entre dispositivos no es requerida y su desarrollo omitible para el propósito. Así pues, dos clases distintas de tipos de peticiones hay que conocer para la correcta ejecución por parte del componente:

- *Non-posted transactions*. Son aquellas transacciones donde el solicitante espera un TLP de respuesta (que podría llegar en uno o varios paquetes) por parte del otro extremo.
- *Posted transactions*. Son otro grupo de transacciones donde no se necesita ninguna clase de confirmación por el otro extremo.

Ejemplos clásicos del primer grupo son escrituras en memoria del anfitrión cuando la petición proviene del dispositivo hardware, mientras que en el segundo encajan las lecturas de memoria principal y su transmisión hacia la FPGA. Se definen en total dos tipos de operaciones a implementar (una operación de cada grupo), cada una de ellas tanto considerando el origen de la petición el *root complex* como el diseño hardware.

- *Non-posted transactions*: Peticiones de lectura de los datos en la FPGA por parte del software (*memory read request*) y lecturas de memoria principal (*memory read request*) por parte del hardware reconfigurable.
- *Posted transactions*. Peticiones de escritura en el espacio de la FPGA por parte del software y escrituras en memoria principal por parte de la FPGA. Ambas operaciones son del tipo *memory write request*.

Con el objetivo de simplificar esta tarea, el *endpoint* presenta 4 interconexiones del tipo AXI4-Stream. Cada una de ellas espera un formato propio, pero lo suficientemente similar a

las cabeceras de un paquete TLP como para que su traducción sea prácticamente inmediata. Las interfaces son:

- *Completer reQuest (CQ)*. Lecturas de la memoria de la FPGA (*Memory read request*), *non-posted operation*, al igual que las escrituras en memoria (*Memory write request*), *posted operations*, por parte del sistema anfitrión son transmitidas sobre la interfaz CQ. Todas estas peticiones vienen de un tercer agente, el software ejecutado sobre la CPU.
- *Completer Completion (CC)*. Los TLPs de respuesta a la interfaz CQ son propagados por esta interfaz. Estas respuestas son generadas por la FPGA.
- *Requester reQuest (RQ)*. TLPs asociados con operaciones *non-posted* tales como la lectura de la memoria del sistema anfitrión (*memory reads*) u operaciones de escritura sobre esa misma memoria (*posted operation*), son transmitidas por esta interfaz. La peticiones son generadas por el diseño de la FPGA.
- *Requester Completion (RQ)*. Paquetes de respuesta a la interfaz RQ son propagados por esta interfaz. Su fuente se localiza en *root* de PCIe (el controlador de memoria).

Así pues, el formato de los distintos paquetes provenientes desde las interfaces asociadas al *completer* quedan reflejados en el Cuadro 4.1 para las peticiones y en el Cuadro 4.2 para los mensajes de contestación. Desde el punto de vista de los paquetes generados por el diseño hardware, el formato queda descrito en el Cuadro 4.3 para las peticiones y en el Cuadro 4.3 se detalla el formato de las respuestas. Aunque la información es muy cercana al formato propio de un TLP, existen ciertas modificaciones aplicadas por el *endpoint*. La descripción detallada de cada uno de los campos se localiza en el Apéndice A.5.

+3 7 6 5 4 3 2 1 0	+2 7 6 5 4 3 2 1 0	+1 7 6 5 4 3 2 1 0	+0 7 6 5 4 3 2 1 0	
Address[31:2]			AT	DWORD 0
Address[63:32]				DWORD 1
Bus	Dev/func	R Req typ	DWORD Count	DWORD 2
R Attr TC	Bar Aperture	BID	Target func / Tag	DWORD 3

Cuadro 4.1: Formato de paquete *memory request* generado por el *root complex* (CQ).

+3								+2								+1								+0								
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
R		L		Byte count												R				AT	R	Address[6:0]												DWORD 0
Bus (REQ)								Dev/func (REQ)					R	P	Sta					DWORD count												DWORD 1
R	Attr			TC		E		Bus (COMP)								Dev/func (COM)								Tag								DWORD 2

Cuadro 4.2: Formato de paquete *memory request completion* generado por el *root complex* (CC).

+3								+2								+1								+0								
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
Address[31:2]																														AT		DWORD 0
Address[63:32]																																DWORD 1
Bus (REQ)								Dev/func (REQ)					P	Req typ						DWORD count												DWORD 2
E	Attr			TC		E		Bus (COMP)								Dev/func (COM)								Tag								DWORD 3

Cuadro 4.3: Formato de paquete *memory request* generado por la FPGA (RQ).

+3								+2								+1								+0								
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
R	C	L		Byte count												Error				Address[11:0]														DWORD 0
Bus (REQ)								Dev/func (REQ)					R	P	Sta					DWORD count												DWORD 1
R	Attr			TC		R		Bus (COMP)								Dev/func (COM)								Tag								DWORD 2

Cuadro 4.4: Formato de paquete *memory request completion* generado por la FPGA (RC).

Hasta este punto se ha visto cómo los datos son recibidos por el *ipcore* asociado a la tarea de abstraer al desarrollador de la transmisión de datos mediante PCIe. Con similitudes a los protocolos usados en una red de ordenadores, diversas capas son presentadas en PCIe. En la capa de nivel de transacción, existen los paquetes son nombrados como TLPs. A su vez, estos son encapsulados en en una capa de enlace y otra capa física como es mostrado en el Cuadro 4.5.

Un paquete TLP encapsulado contiene las siguientes cabeceras desde el punto de vista físico y de enlace (un total de 4 bytes):

3:0 Carácter de inicio 0xF. Usado a modo de palabra mágica y que siempre identifica el inicio de una transmisión.

14:4 Longitud del paquete. Un total de 11 bits que expresan el tamaño en palabras de 4 bytes del paquete.

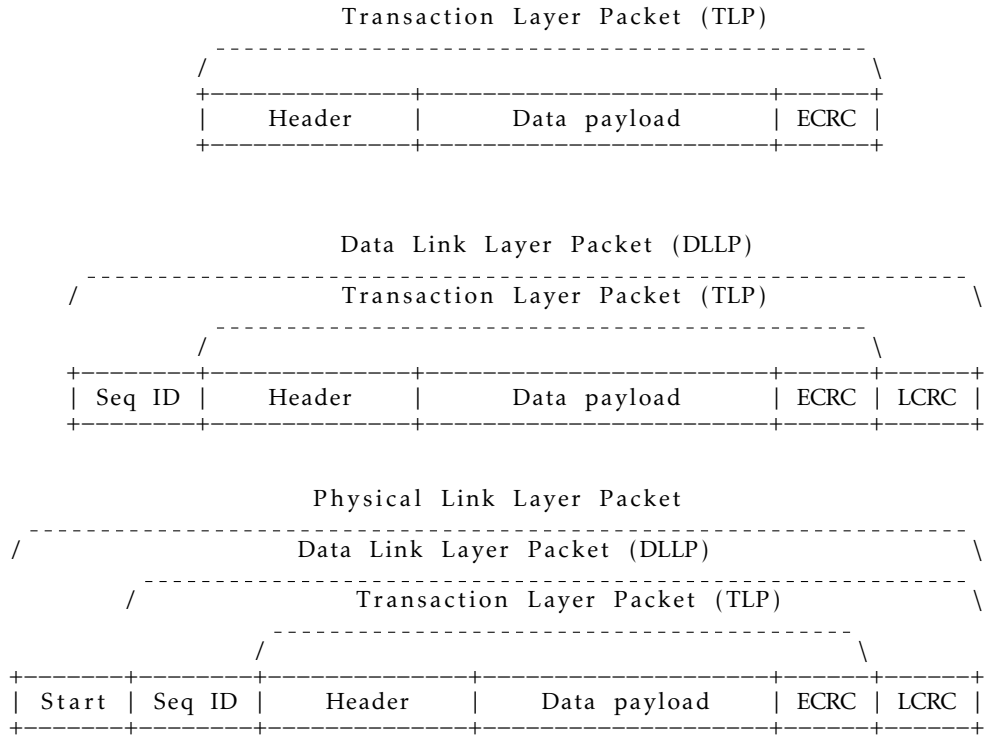
15 Paridad del paquete con objeto de detectar modificaciones producidas por el intercambio

de información entre los enlaces.

19:16 Número de secuencia, bits [11:8].

23:20 CRC de las cabeceras del nivel físico y de enlace.

31:24 Número de secuencia, bits[7:0].



Cuadro 4.5: Capas de un paquete PCIe.

Para un paquete de *request* bajo la capa de transacción se transmite la siguiente información adicionalmente al payload y las cabeceras de niveles inferiores:

10:0 Longitud. Mismo uso que en la interfaz del IP core con la funcionalidad de *endpoint*.

13:12 Atributos. Mismo uso que en la interfaz del IP core con la funcionalidad de *endpoint*.

15 Uso de código de redundancia cíclico adicional (ECRC) al final de paquete.

22:20 *Transaction class*. Mismo uso que en la interfaz del IP core con la funcionalidad de *endpoint*.

28:24 Tipo de paquete.

30:29 Fmt. Usado en conjunción con el campo *type* para indicar la clase de paquete.

35:32 *First Byte enable*. Bytes válidos de la primera DWORD referenciada.

39:36 *Last Byte enable*. Bytes válidos de la última DWORD referenciada.

47:40 Tag.

63:48 *Requester ID*.

127:66 *Address*.

Mientras que para un paquete de respuesta junto al payload (si procede) también se anexiona:

10:0 Longitud.

13:12 Atributos.

15 Uso de código de redundancia cíclico adicional (ECRC) al final de paquete.

22:20 *Transaction class*.

28:24 Tipo de paquete.

30:29 Fmt.

42:32 *Byte count*.

43 BCM. Siempre a 0 salvo cuando un paquete tiene su origen en un bridge con PCI-X

47:44 *Completion Status*.

63:48 *Completer ID*.

70:64 *Address[6:0]*.

79:72 Tag.

96:80 *Requester ID*.

Tras esta breve recopilación de los elementos subyacentes en el intercambio de información a través de PCIe, se pone de manifiesto que hay muchos más elementos involucrados que la simple transmisión del *payload*. Entre los aspectos obligatorios que hay que añadir se localiza un código de redundancia que asegure la integridad a nivel de enlace (LCRC), 4 bytes. El uso de mecanismos para asegurar la integridad adicionales sobre el contenido del paquete (ECRC) es omitible pero posible si la este factor fuera un aspecto crítico. A nivel de paquete TLP, las cabeceras suponen un total de 4 DWORDS al tratarse de operaciones sobre direcciones de 64 bit. Con todo esto se resalta que las comunicaciones se vuelven complejas y la tasa teórica de 8 Gbps por línea nunca es hábil a nivel de datos válidos (no debe olvidarse que a nivel físico se utiliza una codificación 128b/130b de los datos).

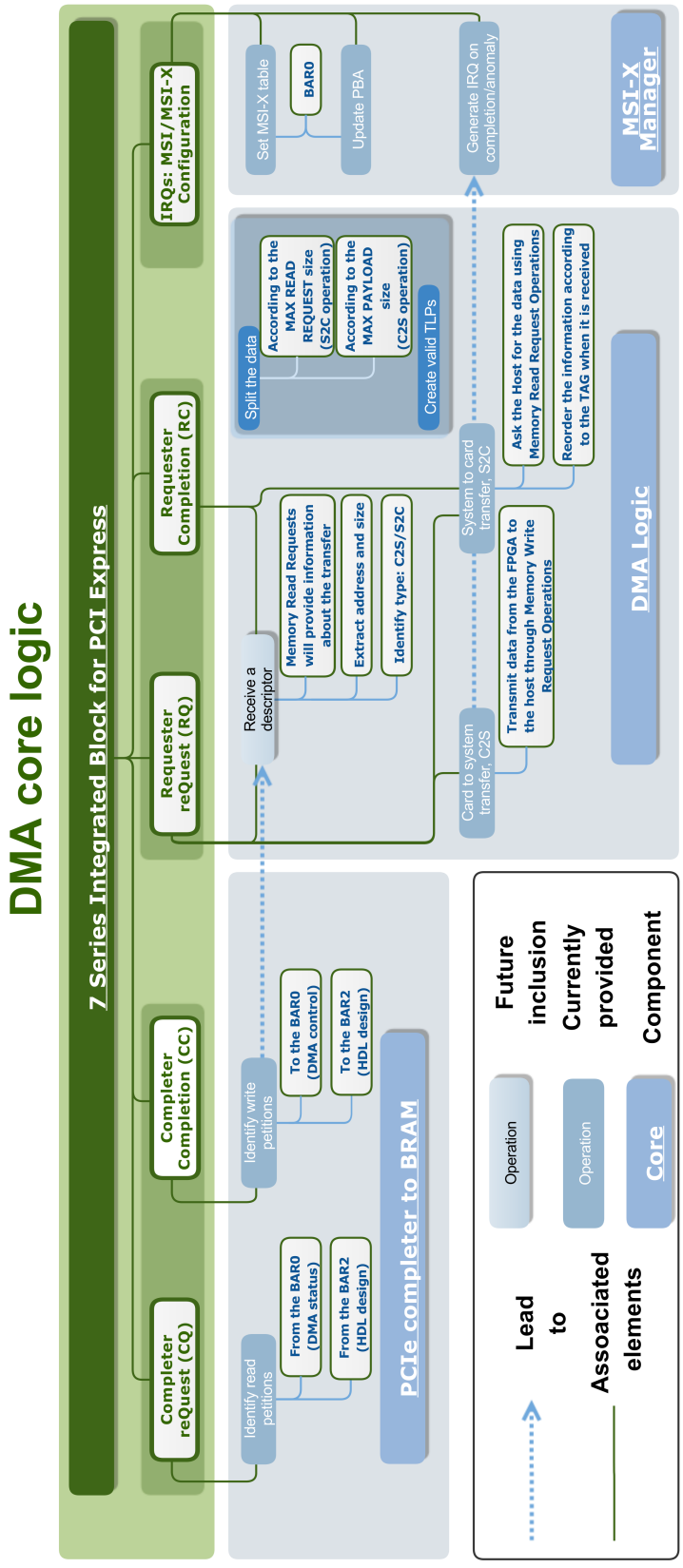


Figura 4.3: Funcionalidades del *core* encargado de las operaciones de DMA.

En la Figura 4.3 se muestra una visión de más alto nivel de las funcionalidades globales del desarrollo y cómo se interactúa con cada una de las interfaces. Cabe destacar la inclusión de una interfaz, dedicada a la gestión de interrupciones por parte del *7 Series FPGAs Integrated Block for PCI Express* no mencionada hasta el momento. En efecto el IP core es capaz de tratar con interrupciones tanto del tipo *legacy* como de tipo message signaled interrupt (MSI)/message signaled interrupt extended (MSI-X) aunque su configuración es completamente opcional. Con la misión de identificar el fin de operaciones, su uso es más recomendable en este tipo particular de operación.

A grandes rasgos, y dejando de momento la implementación a nivel software del *driver* y diseños de usuario, el programa de usuario realizaría una inicialización de los registros del *core* DMA. En dicha configuración se especifica la región de memoria que será participe en la transmisión, número de octetos involucrados y el deseo por parte del usuario al final de la operación de generar una interrupción.

Como recurso adicional, operaciones DMA *scatter-gather* son soportadas. Es decir, la transmisión de una región de memoria no contigua es aceptada tras una previa configuración por parte del diseño basado en CPU. Con estas pinceladas iniciales han aparecido de manera natural dos de los conceptos en los que se sustenta el *core* creado: la figura del descriptor y la del motor.

- Un descriptor es la pieza básica de información que contiene todos los datos necesarios para completar una transacción. Dirección de memoria, tamaño, necesidad de generar interrupción cuando el proceso finalice, estado actual del proceso, tiempo consumido o porcentaje actual son algunos de los campos que se relacionan con un descriptor.

- Un motor es una entidad unidireccional que incorpora una lista circular de descriptores y que permite aplicar transmisiones en una dirección, ya sea en sentido *system to card (S2C)* o *card to system (C2S)*. La lista circular de descriptores se completa mediante dos índices, el puntero al primer descriptor y último hábiles (este último configurable desde el diseño de usuario). En otras palabras, el usuario es capaz de indicar hasta qué descriptor se puede transferir en una determinada operación permitiendo la posibilidad de aplicar *scatter-gather*.

  Como ejemplo supóngase que se desea transferir una región de memoria de 32KB, espacio reservado en 8 páginas (de 4KB) de ubicación no contigua del sistema. El programa que se encarga de generar los datos procesa de manera más lenta que el intercambio de comunicaciones mediante PCIe pero se dispone inicialmente de un bloque de 16KB con información válida. Entonces el usuario dota de valor a 8 descriptores. El primero tiene tamaño 4KB y apunta a la dirección física de la primera página; el segundo tiene el mismo tamaño y apunta a la dirección física de la segunda página y así sucesivamente. De mane-

ra excepcional, los descriptores 3 y 7 (los índices parten de 0) generarán una interrupción en su culminación. El puntero al último descriptor hábil, una vez la configuración esté realizada, se establece a 3 y se habilita la señal de *enable*. En ese momento se comenzará con la transferencia de las 4 primeras páginas. Según los datos se vayan generando, una simple actualización de este índice a 7 (y habilitación de la señal de *enable*) ayudará a que la comunicación del resto de datos sea finalizada.

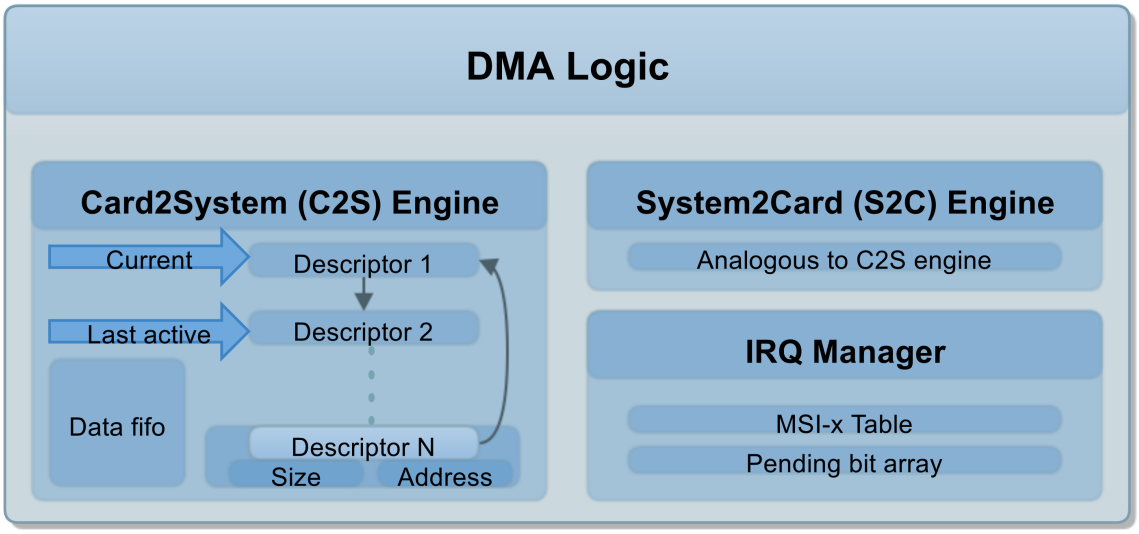


Figura 4.4: Arquitectura simplificada del *core* DMA.

En la Figura 4.4 se muestra como todos los elementos lógicos interactúan entre sí. Nótese que en comparación con la Figura 4.3 se ha dotado de la competencia de administrar las interrupciones al *core* DMA. Esta decisión queda motivada por la necesidad de que cada VF deba implementar su propia tabla MSI-X. En el caso de estudio actual, la cuestión arquitectónica no es realmente crítica pero incluirla dentro del módulo simplificará su tratamiento en el diseño final.

Para resumir, el componente cuenta con dos motores independientes encargados de realizar las operaciones en una única dirección (S2C o C2S). Cada motor cuenta con una FIFO de datos propia donde almacenar la información según es recibida (ya sea por el *endpoint* o desde el diseño hardware) en conjunción con un anillo de descriptores que faciliten operaciones DMA *scatter-gather*.

La problemática del manejo de datos en PCIe

Hasta este punto se ha detallado como interactuar con la lógica de Xilinx *7 Series FPGAs Integrated Block for PCI Express*. Asimismo, la arquitectura y la división lógica en descriptores y motores son conceptos familiares. La encapsulación en un formato fácilmente transformable

a un paquete TLP no es un concepto nuevo pero falta por definir cómo los datos deben ser procesados para su envío/recepción cuando las peticiones provienen desde el propio dispositivo reconfigurable.

En primer lugar, tanto peticiones de *memory read* como de *memory write* están limitadas en el número de bytes transferibles por petición. Las constantes que limitan estas transicciones son conocidas como *maximum payload* (para el caso de *memory write requests*) y *maximum read request* (para el caso de *memory read requests*). El primero suele tomar valores entre los 64 y los 256 bytes; el segundo, desde los 64 bytes hasta los 4096.

Esta configuración es regulable por el sistema operativo por lo que un controlador podría variar su tamaño en cualquier momento siempre y cuando se encuentre dentro de los valores soportados por el dispositivo. Valores más mayores se traducen en un menor *overhead* asociado a las cabeceras y, por ende, un mejor rendimiento global.

El caso menos complicado es aquel donde la operación involucrada es *posted*. Es decir, operaciones de petición de escritura en memoria. Un usuario ha solicitado que le sea transferida cierta cantidad de memoria desde la FPGA hacia una determinada región del sistema anfitrión. En primer lugar, esta información va a ser comunicada en bloques de *maximum payload* bytes. Por cada *maximum payload* bytes debe haber una cabecera (de 128 bits), de acuerdo a la especificada en el Cuadro 4.3. Es necesario tener en consideración que el número de palabras debe ser coherente con lo que se está transfiriendo y que, con casi total seguridad, la última transacción será de un tamaño inferior a *maximum payload*. Es un proceso relativamente asequible pero que involucra trabajar con diseños HDL a altas frecuencias (250.0 MHz).

No obstante, la dificultad real de este proceso se focaliza en las operaciones *non-posted*. Son las peticiones de lectura de memoria. De manera análoga al caso de la operación *posted*, los datos deben ser solicitados en bloques de tamaño máximo *maximum read request*. En estas peticiones no hay carga útil porque *payload* está asociada al paquete de respuesta (Cuadro 4.4). Sin embargo, el campo *tag* juega un papel primordial en este caso.

Tantas peticiones de lectura como *credits* existan son lanzadas simultáneamente. Un crédito expresa el número de transferencias que podrían ser atendidas de manera concurrente. Así pues, si el usuario quiere transferir información a la FPGA, se pedirían los *credits*-primeros bloques de tamaño *maximum read request*.

Con las respuestas pueden ocurrir varias clases de incidencias:

1. La respuesta sea de un tamaño inferior al original. Por tanto, se esperaría recibir en un futuro más paquetes de respuesta asociado al mismo *tag*.

2. Se entremezclen llegando desordenadas. Paquetes con distinto *tag* llegan al *endpoint*.

3. Se reciba un TLP con error.

En el caso de la tercera opción, la operación es cancelada e intenta repetir la solicitud del bloque afectado. Para los otros dos casos, se establece un tamaño determinado (4) a priori. De modo que los *tags* empleados presentarán los valores 0b00, 0b01, 0b10 y 0b11.

Cada *tag* tiene su propia FIFO asociada y se conoce el tamaño total de datos que se esperan recibir ligados a la actual solicitud (típicamente será *maximum read request* salvo para la última petición). Así pues, hasta que el siguiente identificador no ha recibido todos los datos, no se procesará otra petición.

Por ejemplo, supongamos que se han solicitado inicialmente los *tags* 0b00, 0b01, 0b10 y 0b11 en el orden listado. Empiezan a llegar relacionados con cada uno de los *tags* indistintamente hasta que en algún momento se han recibido *maximum read request* bytes asociados al *tag* 0b01. No puede solicitarse otra petición dado el caso de que los datos en la FIFO 0b01 no son los siguientes esperados de manera (se estaría barajando la información puesto que la petición original fue la 0b00). Habrá que mantener ese *tag* inactivo hasta que se completen las respuestas asociadas al *tag* 0b00. En ese instante, tanto la información de la FIFO 0b00 como la de la 0b01 en orden sucesivo puede ser volcada hacia la FIFO propia del motor (facilitando la lectura de los datos por parte del usuario).

Cuando este volcado se ha realizado, nuevas peticiones asociadas a los tags 0b00 y 0b01 pueden ser realizadas pero siempre teniendo en consideración que el siguiente identificador que debería ser exportado es el 0b10.

Al albergar los datos de cada *tag* en una FIFO independiente se permite que si se detecta un error en la transmisión y la interrupción de la actividad, los datos puedan ser descartados y solicitados de nuevo sin afectar a la integridad de la información puesta a disposición del diseño hardware.

En el momento en el que se incorpora soporte para SR-IOV, se dota al *tag* de sentido adicional. Los 2 bits menos significativos (aunque parametrizable en función del tamaño de ventana) se corresponden con el número de la petición actual, exactamente igual que en el caso previo. La diferencia radica en los 6 primeros bits que hacen referencia a la función virtual. Así pues, 0b100 hará referencia al *tag* 0 de la VF 1 o 0b011 especifica el *tag* 3 de la VF 0.

4.1.3 Dotación de capacidades para virtualización a la solución DMA

SR-IOV es una tecnología soportada por el bloque integrado para PCIe. Hasta un máximo de 2 funciones físicas (PFs) y 8 VFs (nunca se podrá exceder este número total de funciones virtuales) son ofertadas. Para la prueba de concepto se emplea una única PF y dos VFs.

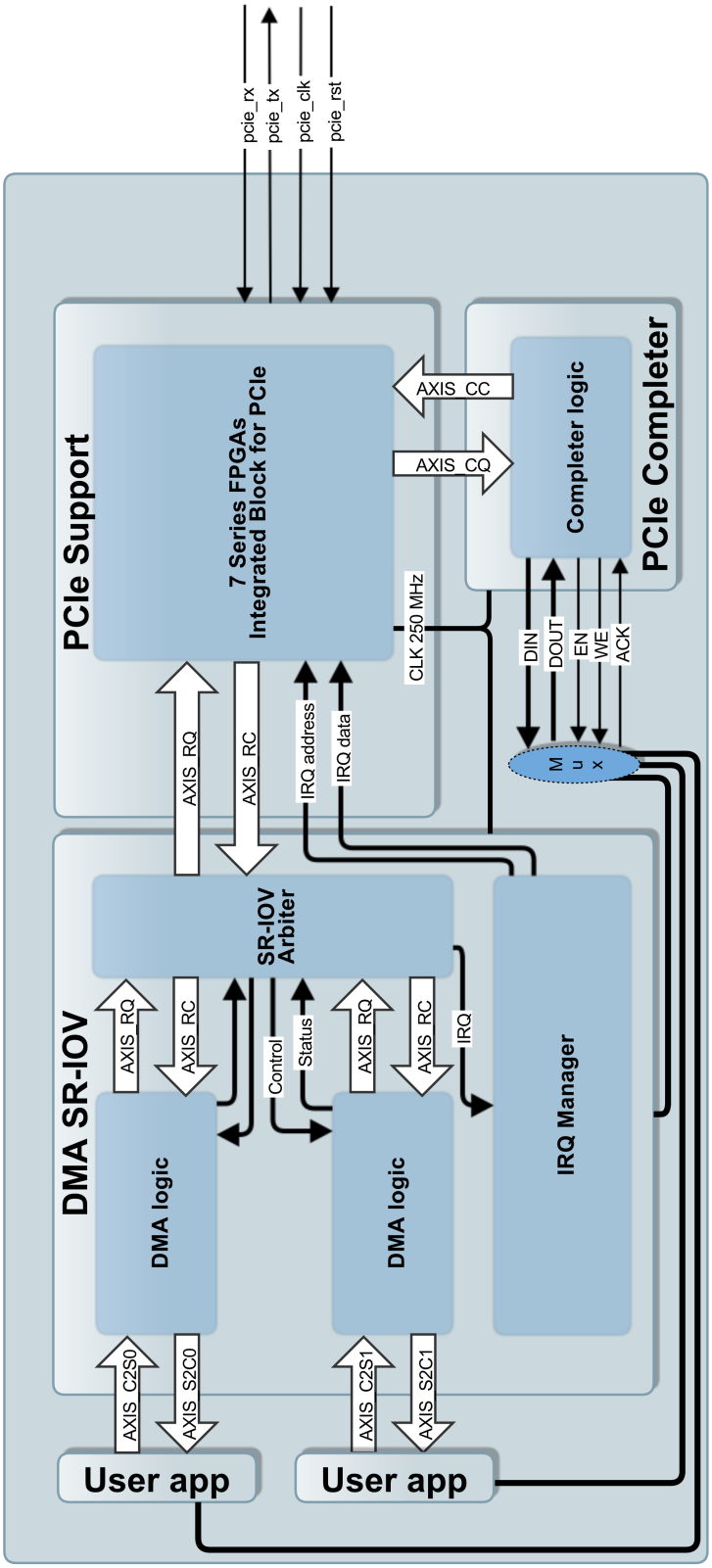


Figura 4.5: Diagrama de bloques de la propuesta de virtualización bajo FPGAs.

Así pues, siguiendo la notación alternate routing ID (ARI), el identificador para las funcionalidades físicas se fija en 0x00 y 0x01 mientras que para el caso de las VF oscila en el rango 0x40-0x47. Para la configuración de una única función física y dos virtuales, los dispositivos son identificados por el índice del bus y las combinaciones 0x00 (PF0), 0x40 (VF0) y 0x41 (VF1).

En la Figura 4.5 se muestra la arquitectura final. Existe la presencia de un módulo encargado de interactuar con las interfaces CQ y CC de manera que se evita tener conocimiento de toda la logística subyacente. Para facilitar esta comunicación se procede a aplicar un cambio de formato con el fin de utilizar una interfaz genérica. El protocolo seleccionado es del tipo memoria. A su vez, en base a la dirección a la que se haga referencia, el origen/destino seleccionado puede corresponderse con el *core* DMA o, por el contrario, con las distintas aplicaciones de usuario desarrolladas a nivel hardware. De este modo, el usuario es capaz de modificar tanto la actuación de las tarjetas de red implementadas en hardware como de las transferencias.

Al mismo tiempo, tantos motores de DMA como funciones virtuales son instanciados. Cada componente genera sus propios valores para los buses AXI4-Stream RQ y RC por lo que un árbitro que seleccione cuál de todos éstos es válido en cada instante del tiempo se precisa necesario. El árbitro desarrollado aplica una política de exclusividad: cuando detecta que uno de los cores necesita transmitir datos, le concede prioridad hasta que termina la operación. Mientras tanto, y en el supuesto de que haya más peticiones de transferencia encoladas, el resto de elementos expectantes deben aguardar su turno.

Esta decisión puede retrasar la entrega de información, pero asegurará una tasa máxima llegado el momento. No debe despreciarse bajo ningún concepto la posibilidad de ejercer otras políticas, como la solicitud simultánea de peticiones. No obstante, es un terreno que no ha sido explorado ni su efecto en el rendimiento general de la aplicación.

Atendiendo a la conectividad de cada *core* DMA se localizan dos interfaces AXI4-Stream que permiten al diseño de usuario transmitir o recibir información sin intervención de la CPU y sin conocimientos de la arquitectura. Únicamente se precisa conocer un estándar como son los buses bajo la especificación AMBA AXI4.

Finalmente, cada motor de DMA puede generar una interrupción si el software, a través de la configuración del descriptor, indicó su obligatoriedad. De este modo, localizando la dirección y dato almacenados en la tabla de vectores, el sistema es capaz de generar una interrupción dirigida a una función particular. En este caso, es el árbitro el que se encarga de detectar a qué tabla MSI-X va dirigida una petición y actuar conforme a las exigencias.

A continuación se detallan los registros y ubicación espacial de cada uno de los mismos en la memoria de la FPGA. El direccionamiento está expresado en palabras de 64 bits y siempre bajo el base address registers (BAR)0 para la configuración de las transacciones:

- Cada función virtual cuenta con su propia tabla MSI-X en el *offset* 0x00 de memoria, mientras que la PBA se localiza en el *offset* 0x100.
- La dirección base para la configuración del *core* DMA asociado a una VF parte de la dirección 0x200. Cada motor (o *engine*) tiene su espacio en múltiplos de 0x100 palabras. En otros términos, dado que un *core* cuenta con dos motores, el primero se configura a partir de la dirección 0x200, mientras que el segundo lo hace a partir de la 0x300. Ambos motores comparten la misma estructura:

0x200/0x300 Registro de control (lectura y escritura, RW).

- 0 *Enable* (RW). El bit 0 se corresponde con la señal de *enable*.
- 1 *Reset* (RW). El bit 1 (activo a nivel alto) fuerza un reinicio del estado actual del *core*
- 2 *Running* (sólo lectura, RO). Indica si este motor está pendiente de finalizar una operación que fue asentida por el usuario (*enable* fue establecido a 1).
- 3 *Stop* (RO). Indica si este motor está parado por haber completado una operación.
- 4 *Error* (RO). Indica si durante una operación se ha detectado alguna clase de error no recuperable.
- 6:5 *Capabilities* (RO). Indica la dirección de operación del motor. 0b01 indica un motor encargado de operar en la dirección *c2s*, 0b10 indica un motor que opera en la dirección *s2c*. Se utilizan dos bits por posibles mejoras donde un motor pudiera presentar capacidades bidireccionales. En el momento de redacción del presente documento, los motores pares se corresponden con aquellos capaces de operar sobre *c2s* mientras que los impares cumplen con la funcionalidad recíproca.
- 7 *IRQ pending* (RO). Indica si la operación ha terminado y se ha generado una interrupción que está pendiente de ser procesada.

0x201/0x301 Puntero al último descriptor configurado hábil (RW).

- 7:0 Índice al último descriptor que puede ser tomado en consideración y ejecutado por parte de la lógica de transferencia.

0x202/0x302 Tiempo (RO). Número de ciclos de reloj, a la frecuencia de 250.0 MHz, necesarios para completar la anterior petición por parte del usuario en el motor.

0x203/0x303 Tamaño de la transferencia (RO). Número de octetos completados como parte de la anterior solicitud del usuario.

– Espacio de configuración del descriptor 0.

0x204/0x304 Dirección (RW). Puntero de 64 bits a la región de memoria sobre la que se almacenará/leerá la información de la transferencia.

0x205/0x305 Tamaño (RW). Entero de 64 bits con información sobre el número de bytes a transferir desde/hacia la región referenciada por el campo *dirección*.

0x206/0x306 Registro de control (RW).

0 IRQ. Este bit indica si a la finalización de la transferencia del descriptor 0 deberá generarse una interrupción.

– Espacio de configuración del descriptor i para $i = 0, ..., ndescriptors - 1$, motor j para $j = 0, 1$.

$0x100(j+2)+3i+4$ Dirección (RW). Puntero de 64 bits a la región memoria sobre la que se almacenará/leerá la información de la transferencia.

$0x100(j+2)+3i+5$ Tamaño (RW). Entero de 64 bits con información sobre el número de bytes a transferir desde/hacia la región referenciada por el campo *dirección*.

$0x100(j+2)+3i+6$ Registro de control (RW).

0 IRQ. Este bit indica si a la finalización de la transferencia del descriptor i deberá generarse una interrupción.

De este modo, se verifica que cada una de las funciones virtuales tiene su propio espacio de configuración, aislado por el propio diseño hardware del resto. Se asegura la coherencia de las transacciones entre las distintas VFs a nivel de hardware, evitando complejos mecanismos de sincronización entre VMs.

4.2 Controladores físicos y virtuales

Para completar la sección se procede a detallar las características del software controlador para entornos Linux. Ninguna configuración adicional debe presentar la máquina virtual salvo que su kernel debe estar comprendido entre las versiones 2.6.32 y 4.0.9 y disponer de los paquetes kernel-dev y kernel-headers instalados para ser capaz de crear el compilado del módulo controlador.

De manera similar, el controlador de PFs, al necesitar habilitar las funciones virtuales asociadas a la misma, requiere de un núcleo más moderno. Se sitúa el límite inferior en la revisión 3.2.70 aunque para versiones anteriores, podría llegarse a adaptar si fuera estrictamente necesario.

A pesar de ello, se ha buscado elaborar un controlador sencillo arquitectónicamente pero con la totalidad de la funcionalidad que pudiera necesitarse en un entorno de producción. En cualquier caso, diferencias entre la versión asociada a la función virtual y a la función física varían notoriamente.

4.2.1 Controlador físico

El módulo controlador ligado a una PF carece de posibilidad de configuración por parte del usuario. Las características son rígidas y no alterables. No existe necesidad de comunicación con ningún otro proceso y/o controlador. Asociadas a este *driver* se distinguen las siguientes tareas: instanciación en memoria, extracción, *probe* del dispositivo y eliminación del mismo.

- La rutina de instanciación es ejecutada cuando se carga el módulo mediante un comando como *insmod*. Se especifican el *device* y *vendor* del dispositivo a auditar, así como las funciones a invocar cuando éste se conecte, suspenda o, en líneas generales, cambie de estado.
- La rutina de extracción del dispositivo tiene la misión de dejar el sistema intacto tomando como modelo de referencia el instante de tiempo en el que la función de instanciación fue invocada por primera vez. Este proceso de limpieza aparece tras la invocación del comando *rmmod* por parte del administrador del sistema.
- La rutina *probe* es la encargada de reservar los recursos para la PF una vez que se detecta el dispositivo con *vendor* 0x10EE y *device* 0x7038. En este caso particular únicamente habilita las dos VFs asociadas a la función física y se establece tanto *maximum read request* como *maximum payload* al máximo valor soportado por el dispositivo hardware.
- Finalmente, la rutina de eliminación deshabilita las funciones virtuales y libera el recurso PCIe.

No es necesario añadir mucha más información a este *driver* debido a que se limita a habilitar las VFs y, por tanto, su lógica es reducida.

4.2.2 Controlador virtual

El controlador asociado a una VF está completamente aislado de la situación. Es decir, el mismo *driver* originario que pudiera usarse para la comunicación DMA se puede reaprovechar en esta ocasión. Sin embargo, al tratarse de una plataforma híbrida, donde tanto desarrollo hardware como software coexisten, comunicación con aplicaciones ejecutadas en el entorno de usuario son requeridas.

De modo análogo al controlador físico funciones para la instanciación (modificando el *device* por el valor adecuado, 0x7028), extracción, sondeo del dispositivo y eliminación también se encuentran presentes. Sin embargo, en la etapa de sondeo (y, por tanto en su liberación) un número mayor de recursos debe ser reservado. Se destacan las siguientes características:

- Manejo de un dispositivo de caracteres para la comunicación directa con la FPGA. Gracias a las instrucciones *input/output control (IOCTL)* propias definidas se permite realizar dicha tarea.
- Manejo de un fichero virtual (reporte en la carpeta *proc* del sistema) para visualización de las estadísticas.
- Implementación de las rutinas manejadoras de interrupciones. Cada componente DMA es capaz de generar dos interrupciones diferenciadas: uno asociada a cada motor.

Descripción en lenguaje natural del módulo kernel para VFs

```
Instancia:
  Solicitar recurso con VENDOR=0x10EE y DEVICE=0x7028.
  Pedir memoria y recursos sistema (semáforos y colas de trabajo).
  Habilitar PCIe.
  Realizar el map de los distintos base address register.
  Habilitar char device y crear estructura bajo /proc.
  Habilitar interrupciones.
-------------------------------------------------------------------------------
Ejecución:
  Si hay IOCTL:
        Si registrar buffer:
                Convertir la dirección virtual a dirección de bus. Guardar la dirección
    de inicio de cada página.
        fin si
        Si liberar buffer:
                Esperar a terminar las operaciones activas.
                Olvidar las direcciones de bus.
        fin si
        Si operación C2S:
                Configurar descriptores en dirección hacia el anfitrión.
                Habilitar interrupción en el último descriptor.
                Actualizar bit enable.

                Dormir en semáforo hasta que la rutina manejadora nos despierte.
                Actualizar datos sobre tamaño recibidos.
        fin si
        Si óoperacin S2C:
                Configurar descriptores en dirección hacia la tarjeta.
                Habilitar interrupción en el último descriptor si el usuario lo indica.
                Actualizar bit enable.

                Dormir en semáforo hasta que la función manejadora nos despierte (sólo
    si el usuario lo indica).
        fin si
  fin si
```

De este modo hasta un total de 8 operaciones IOCTL son implementadas: dos asociadas con la manipulación de las zonas de memoria sobre las que se transfiere la información y seis ligadas a operaciones transferencia, tanto mediante DMA (una por cada dirección a modo de petición y otras dos para esperar a la generación de interrupción) como por intervención directa de la CPU.

Compartición de memoria entre espacio de usuario, sistema operativo e invitados

El uso de páginas de tamaño no estándar (páginas de tamaño superior a 4KB tradicionalmente), no ha sido inicialmente previsto para su implantación a nivel de kernel en entornos linux. Su aplicación en herramientas que se ven obligadas a manejar una cantidad masiva de datos, como pudiera ser un gestor de base de datos, es algo ampliamente desarrollado y estudiado mas no tanto su empleo desde un módulo [Edward Whalen, 2013]. La explicación de la mejora en rendimiento está justificada gracias al principio de localidad espacial, efectos de memorias cachés en el acceso a datos, el menor nivel de carga de la *Memory Management Unit (MMU)* y la imposibilidad de aplicar *swap* sobre estas regiones. No obstante, estas regiones de memoria reservadas en el momento de arranque del sistema, privan al mismo de la posibilidad de reutilizarlas para otra clase de fines cuando no estuvieran formando parte activa del sistema.

Así pues, para un número intensivo de transferencias, el acceso a regiones contiguas contribuiría con un mayor rendimiento. Este es el entorno donde la arquitectura obtendría una mayor ventaja. Se procede con la implantación de páginas no estándar en el desarrollo del *driver*. Se limita el grado de disociación entre módulo y nivel de usuario, ya que el programa ejecutado en el entorno menos privilegiado será quien reserve estas páginas (a través de una capa intermedia del sistema operativo, "middleware") y el *driver* está obligado a conocer que la actividad será ejecutada de esta manera.

La propuesta de arquitectura, comprende el uso de *huge pages* tanto para la ejecución de la máquina virtual invitada como para la reserva de las regiones de transferencia. El sistema operativo anfitrión reserva tantas páginas de tamaño no convencional (y del mayor posible, 1GB) como fuera posible sin comprometer el entorno. Cada VM se ejecutaría sobre estas regiones para asegurar un mayor rendimiento. A su vez, el sistema operativo invitado reservaría *huge pages*, ligadas a regiones de las páginas del sistema original pero esta vez de un tamaño menor (2MB). Con este planteamiento, las regiones a transferir nunca serán volcadas a memoria secundaria a la vez que se asegura que las operaciones de DMA puedan ser lo mayor posibles.

Programas a nivel de usuario: desarrollo *loopback*

El diseño final a nivel de usuario queda completamente abstraído de toda la arquitectura subyacente, salvo por el detalle de que la comunicación con el *driver* virtual debe existir en lugar de utilizar las funcionalidades propias ofertadas por el sistema operativo.

Así pues, una aplicación de referencia encargada de monitorizar el tráfico de red en modo bucle, es decir, escuchar por una interfaz lo mismo que se está enviando por otra se representa a continuación:

Descripción en lenguaje natural de diseño de usuario

```
Reserva de n páginas de memoria.
IOCTL(Registrar buffer, dirección virtual n páginas)


mientras( noFinDeTraza() )
        i = j  mód n/2 páginas
        IOCTL(operación S2C, página i, tamaño huge page)
        página i ← leerDatosTraza()
        IOCTL(memoria libre, página i, tamaño huge page) # Esperar a S2C IRQ
                                    # que involucra la página i  si procede

        IOCTL(operación C2S, página i+n/2, tamaño huge page) # Operación asíncrona
        IOCTL(esperar C2S) # Esperar dormidos en semáforo a nivel de driver

        si( comp(página i+n/2, página i )  no es VERDADERO )
                salir error
        end si
        inc j
fin mientras


IOCTL(Liberar buffer, dirección virtual n páginas)
Liberar n páginas
```

Leyendo de fichero hasta el final del mismo, se guarda la información en las primeras $\frac{n}{2}$ páginas, mientras que los resultados leídos se guardan en las $\frac{n}{2}$ segundas. De este modo, los datos son comparables en búsqueda de posibles errores. No obstante, en todo el proceso existe una capa de *middleware* encargada de encapsular las llamadas directas al *driver* para evitar posibles variaciones ante el cambio de la interfaz. Por sencillez a la hora de comprender el código, y con objeto de entender los puntos de contacto con el *driver*, se ha plasmado la versión menos abstraída de la implementación del módulo controlador.

5 Experimentos y resultados

Hasta el momento actual se han explorado las distintas características que un entorno de desarrollo con capacidad para virtualizar un elemento hardware debe otorgar. Es importante tener en mente, que no es el rendimiento el principal objetivo de este trabajo pero que es una medida primordial para muchas de las aplicaciones que requieren de unidades de coprocesamiento.

Así pues, con objeto de probar la nueva funcionalidad de transferencia de DMA desarrollada, se plantea un escenario donde se explotan todos los recursos pero a la vez ofrece un tiempo de desarrollo asequible: transmisión de enteros sucesivos, tanto desde la FPGA como desde la CPU, de modo que el otro extremo se cerciore de que los datos recibidos son correctos. De esta manera se independiza la funcionalidad de transferencia de la lógica de comunicación en redes de ordenadores.

5.1 Tarjeta de red de altas prestaciones: 10 Gbps

Para hablar de prestaciones a nivel de tarjeta de red debe hacerse una evaluación de todos y cada uno de los componentes de la cadena de procesamiento. En primer lugar, se debe distinguir entre los distintos elementos involucrados en todo el proceso de transmisión/recepción de paquetes cuáles suponen realmente un posible cuello de botella. Por ejemplo, las interfaces de red aseguran una tasa física de 10 Gbps. Es decir, no es viable obtener ningún rendimiento superior a este valor sobre la plataforma objetivo. No depende del diseño del programador mejorar su rendimiento dado que es una limitación ajena a él.

En cualquier caso, sería deseable que no se experimentase ninguna penalización a esa tasa a lo largo de todo el proceso. Cada uno de los elementos involucrados en el sistema deben ser capaces de trabajar a una tasa mayor o igual a la ya citada. Realicemos un análisis de los distinguidos elementos:

(a) Componentes de terceros, IP cores. Tanto el diseño *10 Gigabit Ethernet PCS/PMA (10GBASE-R)* como el *10 Gigabit Ethernet Media Access Controller* están planificados para trabajar a tasa de línea en enlaces multigigabit de 10 Gbps. Esta afirmación se traduce en que se es capaz de trabajar con hasta un total de 14880952 paquetes por segundo, introduciendo una ligera latencia en el procesamiento de la información pero poniendo a disposición del resto de elementos del diseño la posibilidad de trabajar con el enlace a pleno rendimiento. En el caso de la tarea de recepción, es a la salida del *core* MAC donde se permite

aplicar el marcado temporal de los paquetes con una resolución del orden de nanosegundos mejorando la precisión de cualquier marcado software.

(b) Diseños en lenguajes HDL orientados a la generación de una tarjeta de red. El módulo que caracteriza a la aplicación en cuestión es la conversión de la señal del protocolo *Attachment Unit Interface (XAUI)* a paquetes TLP que sean transferidos al anfitrión (y viceversa). Dada la reutilización de componentes (*core* DMA e IP cores de terceros) con su consecuente abstracción en la elaboración final del sistema, la lógica restante se limita casi en su totalidad a la interconexión entre módulos.

En este recorrido las dos únicas posibles limitaciones se tratan de la memoria utilizada para el cruce de dominio de reloj y la política seleccionada para la asignación entre paquetes e interfaces (en transmisión). Se persigue que ninguna de ellas imponga una penalización a la tasa global.

Para garantizar el primer escenario se utilizan memorias de acceso rápido (BRAMs). Mientras que una memoria de acceso aleatorio (RAM) síncrona no puede leer, modificar o escribir en un único ciclo de reloj, las FPGAs de Xilinx que cuentan con BRAMs pueden aplicar *pipeline* sobre las operaciones de escritura para obtener un rendimiento de una lectura/escritura/modificación por ciclo en cada puerto.

Para el segundo caso, una política de turnos (*Round Robin*) puede ser suficiente (no depende del valor tomado por los datos) y sencilla de implementar. De manera alternativa, si se quisiera repetir el tráfico por varias interfaces, una máscara sobre salidas habilitadas podría ser tenida en cuenta. La decisión de qué planificador es más conveniente varía profundamente según la tarea particular que deba cumplir la tarjeta de red. Sin embargo, bajo ningún concepto deben perderse de vista las limitaciones sobre el rendimiento global que puede acarrear la selección de la interfaz, dado que añadir retardos innecesarios podría disminuir el número de paquetes concurrentes procesados.

Como conclusión, la generación de una tarjeta de red capaz de trabajar a tasa de línea a 10 Gbps es una realidad y una tarea viable, en cuanto a tiempo de desarrollo se refiere, si se emplean componentes de terceros (o se sustenta parte del desarrollo en la reutilización de código). No debe perderse de vista, en ningún caso, de las capacidades de cómputo de una FPGA. Así pues, filtros basados en firmas pueden ser implementados en hardware ahorrando la tarea de extracción y clasificación. Generalmente, suponen un trabajo no despreciable para la CPU [Zazo et al., 2015] aunque esta clase de técnicas excederían a las exigencias de un primer acercamiento a una tarjeta de red.

5.2 Transferencias mediante DMA

Para la ejecución del experimento se cuenta con un equipo con placa base Supermicro X9DRD-iF, dos procesadores Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, 64 GiB DDR3 RAM de memoria a 1600 MHz (8 bancos de 8 GiB, 4 módulos conectados a cada CPU) y la placa VC709 de Xilinx (que incluye una FPGA Virtex-7 XC7VX690T-2FFG1761C) conectada directamente a la primera CPU.

La afinidad a nivel de hilo del sistema es contemplada en todo momento, de modo que la segunda CPU quede aislada siempre de las operaciones de intercambio de información. Exclusivamente una VM y una VF son creadas bajo el hipervisor máquina virtual basada en el kernel de Linux (KVM). Para el experimento la VM contará con un total de 8 GiB de memoria RAM y un máximo de 4 *cores* de la CPU. Tanto VT-x como VT-d y SR-IOV están habilitados tanto a nivel de BIOS como a nivel del sistema operativo anfitrión. Ambos sistemas utilizan entornos Linux: Ubuntu Server 14.04 (kernel 3.16) en el caso de la estación anfitriona, mientras que el caso del sistema invitado es un CentOS 7 (kernel 3.10).

Se subdividen las mediciones en dos categorías para facilitar la detección de comportamientos similares en el proceso:

- Prueba del rendimiento en sentido C2S. En este caso, el diseño de usuario de la FPGA genera números enteros que son puestos a disposición del software mediante transferencias DMA. Es en este último nivel donde se verifica la integridad de los datos recibidos.
- Prueba del rendimiento en sentido S2C. A nivel de software, en el espacio de usuario, se inicializa una región de memoria con números enteros consecutivos que serán transferidos mediante DMA. Es el diseño hardware el que se encarga de su validación una vez que la operación ha sido completada.

A su vez, con el fin de medir el rendimiento, se repite la prueba en distintas situaciones:

(a) Usando el diseño del *core* DMA sin soporte para virtualización (*native*).

(b) Realizando las operaciones de transferencia sobre la PF creada.

(c) Realizando las operaciones de transferencia sobre la VF creada pero conectada al equipo anfitrión (sin *overhead* de máquina virtual).

(d) Realizando las operaciones de transferencia sobre la VF creada pero conectada mediante *PCI passthrough* al sistema invitado.

Sin embargo, antes de proceder a mostrar los resultados es conveniente prestar atención a las estimaciones teóricas esperables del proceso. PCIe 3.0 oferta un ancho de banda de 8 Gbps por línea. En este caso, existen un total de 8 líneas y, por ende, un ancho de banda teórico de 64 Gbps es alcanzable. Sin embargo, debido a la codificación física (128b/130b) y al *overhead* de las cabeceras de un paquete TLP esta cuantía disminuye. Un supremo en la dirección C2S es fácilmente estimable considerando que el tamaño de la cabecera de un paquete transmitido mediante PCIe son 196 bits (4 bytes del medio de enlace, 16 bytes asociados a una cabecera TLP para direcciones de 64 bits y, finalmente, otros 4 bytes del LCRC). Así pues, la cota máxima para el rendimiento viene dada por:

$$64 * 10^9 \frac{b}{s} * \frac{128}{130} * \frac{MAX\_PAYLOAD * 8}{MAX\_PAYLOAD * 8 + 192}$$

Lo que se traduce en una tasa máxima de 57.61 Gbps en el caso en el que tamaño máximo del *payload* soportado sea 256 bytes o 53.06 Gbps si exclusivamente 128 bytes es la mayor capacidad soportada por el sistema. La dirección S2C será típicamente más crítica en el sentido de que ante un TLP de solicitud debe esperarse una respuesta del *complex root* trasladando la información. Es por ello que no se esperaría ver una velocidad de transmisión mayor.

En la Tabla 5.1 se muestra el porcentaje ocupado por el diseño mientras que en las Figuras 5.3 y 5.4 se representan los resultados de rendimiento conjuntos. La tendencia está bastante clara: para pequeñas cantidades de datos no está claramente justificado el uso de operaciones de DMA. Por cantidades pequeñas, basándose en los resultados empíricos, podrían ser consideradas aquellas inferiores a los 256KB aproximadamente de acuerdo con las Figuras 5.1 y 5.2. Después de este valor el rendimiento parece estabilizarse ofreciendo unos resultados bastante prometedores.

Lógica	**Diseño DMA**	**Diseño soporte SR-IOV**
FF	8,784 (1,03%)	8,951 (1,05%)
LUTs	9,171 (2.13%)	9,238 (2.13%)
Memory LUTs	136 (0.08%)	136 (0.08%)
Block RAMs	19 (1.29%)	19 (1.29%)
BUFGs	5 (15.62%)	5 (15.62%)

Tabla 5.1: Sumario de utilización del dispositivo.

En la dirección C2S una media de 51.74 Gbps es alcanzada cuando se copia una región de 1MB. Para este mismo tamaño de transferencia, 50.40 Gbps es la tasa asegurada en la dirección S2C para la versión del *core* sin funcionalidad de virtualización. Para la recopilación de estos datos, se ha contabilizado el tiempo desde el instante previo a la configuración de los

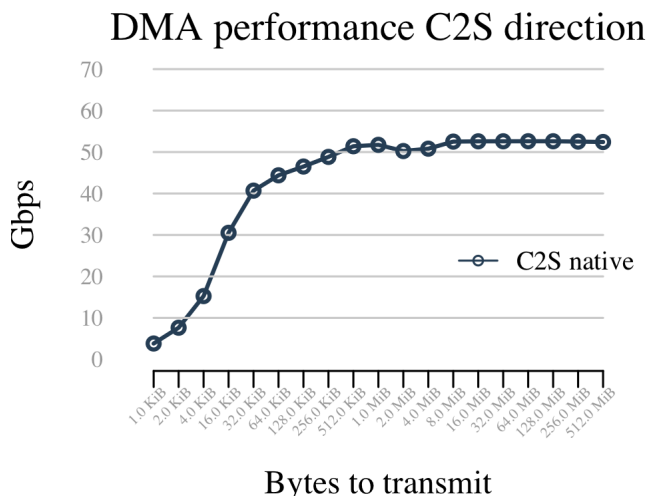


Figura 5.1: Rendimiento nativo en transferencias DMA en la dirección C2S.

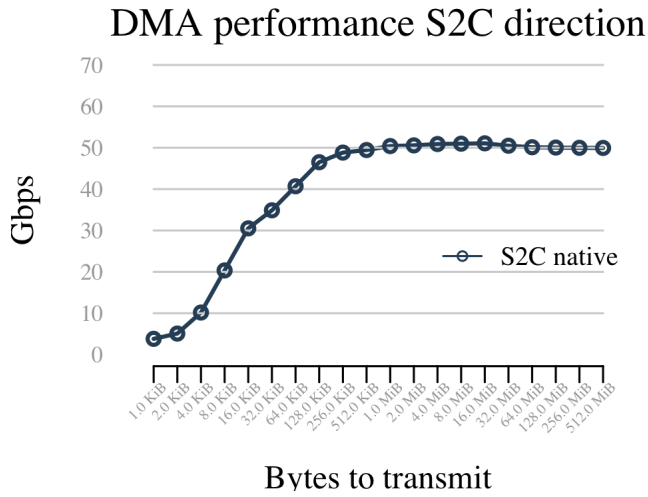


Figura 5.2: Rendimiento nativo en transferencias DMA en la dirección S2C.

descriptores en el diseño hardware. De tal modo la estimación teórica nunca será alcanzable, justificando la caída respecto al máximo valor teórico esperable.

Si se presta atención a la dirección C2S, el comportamiento es bastante similar en cada tecnología. Transferencias que involucran regiones mayores de 128KB aseguran al menos el 90% del rendimiento de la versión nativa, o sin soporte para virtualización. Sin embargo, no debe olvidarse que dado el caso en el que múltiples VMs quieren transmitir datos simultáneamente, la transferencia en alguna de ellas podría ser demorada.

No obstante, los resultados en la dirección S2C no son tan alentadores. Del valor de referencia para la versión nativa de 50.40 Gbps en transferencias de 1MB, una VM haciendo uso de la VF únicamente ha sido capaz de alcanzar una tasa de 6.00 Gbps (11.90% del caso nativo). La justificación a este incidente reside en el factor de que el uso SR-IOV en la arquitectura está

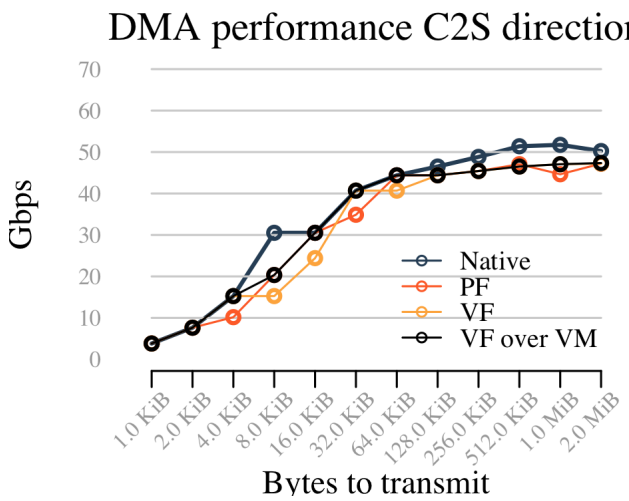


Figura 5.3: Rendimiento ofrecido por el core DMA en la dirección C2S.

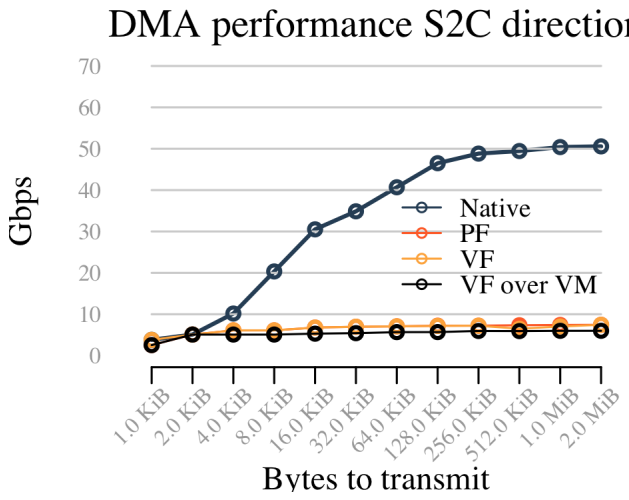


Figura 5.4: Rendimiento ofrecido por el core DMA en la dirección S2C.

limitando el *maximum read request* y *maximum payload* a 128 bytes. Este hecho involucra que las operaciones C2S han descendido de un tamaño original de 256 bytes de *maximum payload* a la mitad. Aunque es una diferencia considerable, no es tan loable como la apreciada en las transferencias S2C, donde se ha pasado de un valor original de 4096 bytes para el campo *maximum read request* a una treintadosava parte. Esta limitación acentúa una caída del rendimiento en la transferencia de datos desde la memoria principal a la FPGA.

Se trata de un problema ajeno al proyecto, limitado por la arquitectura hardware subyacente en el sistema anfitrión. Al tratarse de tecnologías de reciente difusión e interés por la comunidad, el soporte muchas veces es escaso y, como ha ocurrido en esta ocasión, no plenamente satisfactorio. Sería de esperar que los fabricantes de hardware (placas base) mejoren su soporte para SR-IOV, permitiendo, de ese modo, alcanzar unas cifras bastante parejas a las nativas que permitan a un grupo mayor de aplicaciones ser susceptibles de virtualización.

6 Conclusiones y trabajo futuro

A lo largo del presente texto se ha presentado una arquitectura sustentada en hardware reconfigurable que permita la virtualización de un dispositivo prohibitivo económicamente como para tener tantos como VMs. Aunque ejemplificado como una tarjeta de red multigigabit, las posibilidades abarcan un espectro mayor, desde unidades de coprocesamiento hardware hasta elementos de procesamiento digital de señales. El gran hito alcanzado ha sido probar que toda esta tecnología es viable, adaptable a una gran cantidad de problemas y, en última instancia, la distribución de un entorno que permita a otros usuarios disfrutar de la virtualización sin tener que descender a estos niveles de detalles. Otros logros marcados también han sido:

- Utilización de una plataforma accesible para el mundo empresarial.
- Incorporación de sistemas operativos libres en el proyecto que puedan ser extendidos fácilmente para la consecución de los objetivos particulares. A su vez, muchas veces las licencias de estos sistemas también son gratuitas eximiendo de un gasto no despreciable.
- Contribución a la comunidad del *free software/hardware* con un *core* DMA de altas prestaciones con vistas a trabajar en redes multigigabit de 10 Gbps y 40 Gbps.
- Al no rescindirse a un modelo concreto (salvo por la familia de FPGAs empleada), esta plataforma puede ser sustentada por un entorno simplificado tanto en características como en coste para que el precio no suponga realmente una barrera respecto a la compra de una tarjeta de red convencional (o del dispositivo particular implementado en hardware).
- Aislamiento de gran parte del cómputo en la CPU al desarrollar tecnologías que liberan de carga de trabajo a la misma. Desde las operaciones de transferencia de datos DMA hasta la inclusión de interrupciones MSI-X permiten que el rendimiento sea idóneo.
- Investigación y aporte a la comunidad científica sobre el estado de SR-IOV en entornos de alto rendimiento.

Estos aspectos han desembocado en el desarrollo de una arquitectura funcional y válida para su utilización por terceros. Todos los elementos que integran el proyecto final han sido probados en situaciones reales, ofreciendo una estabilidad mucho mayor que aquellas soluciones exclusivamente simuladas. Independientemente, se ha conseguido tanto la elaboración de un motor de DMA funcional como una tarjeta de red al uso bajo el *core* de *Northwest Logic*. La creación de una tarjeta de red que emplee el nuevo entorno está en proceso pero no había sido objeto de interés hasta el momento, con la liberación final de la plataforma.

Y es que los resultados obtenidos en el presente trabajo han causado una grata impresión en proyectos independientes a la universidad, como NetFPGA [Gibb et al., 2008] en la Universidad de Cambridge, donde han mostrado su predisposición a adherir la tecnología ofertada. Se espera su distribución en la última versión del proyecto, NetFPGA Sume, en los próximos meses ya que la encapsulación y la adaptación a la plataforma libre está actualmente en proceso.

No obstante, existe un duro camino por delante a la hora de seguir explotando la tecnología de virtualización dado que ni mucho menos, todo ha quedado dictaminado en esta aproximación. Desde el estudio de algoritmos de balanceo de carga a la hora de transmitir paquetes de red por las interfaces de una tarjeta, hasta cómo lidiar con máquinas virtuales que desean transferir concurrentemente (no debe olvidarse que actualmente está implementado como un recurso exclusivo, es decir, hasta que no libera la comunicación un motor, no es aprovechable por otro). Todo esto sin olvidar los problemas de rendimiento en transferencia de datos desde el hardware reconfigurable.

En líneas generales, este trabajo ha posibilitado desmentir la creencia popular de que virtualización siempre está reñida con el alto rendimiento. En un mundo donde cada vez más la computación está orientada a grandes *mainframes*, la posibilidad de que un usuario particular pueda aprovechar un recurso hardware reconfigurable para sí mismo es una gran ventaja. Exploración sobre la reconfiguración parcial y SR-IOV son viables [Vu et al., 2014] y favorecerían una nueva manera de entender las FPGAs bajo las nuevas tendencias de procesamiento distribuido. Con vistas a un trabajo futuro, por tanto, se presentan las siguientes cuestiones:

- Colaboración con el proyecto NetFPGA Sume para la inclusión de la tecnología de virtualización a nivel hardware en su iniciativa de libre distribución.
- Seguir trabajando en mejoras a nivel de desarrollo hardware: descriptores en memoria, soporte para un número mayor de funciones virtuales/físicas, etc.
- Planteamiento de aplicación de alto rendimiento que explote estas características demostrando la sencillez y flexibilidad que dota al sistema usar VMs. Por ejemplo, monitorización del tráfico de red. Cada interfaz puede ser monitorizada por una única VM. Una vez creado el diseño para una interfaz de red, la repetición de componentes lógicos permitirían realizar este proceso sin tener que aplicar apenas modificaciones en niveles superiores.
- Intercalar el concepto de virtualización con reconfiguración parcial, permitiendo a un gran hardware reconfigurable, con prestaciones muchas veces infrautilizadas, ser explotado adecuadamente. Esta idea guarda gran relación conceptualmente con la utilización de máquinas virtuales sobre un sistema anfitrión que no es totalmente exprimido.

Bibliografía

[Abbott, 2000] Abbott, D. (2000). *PCI Bus Demystified*. L L H Technology Publishing.

[Attig y Brebner, 2011] Attig, M. y Brebner, G. (2011). 400 gb/s programmable packet parsing on a single fpga. En *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, páginas 12–23.

[Bellard, 2005] Bellard, F. (2005). QEMU, a Fast and Portable Dynamic Translator. En *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, páginas 41–41, Berkeley, CA, USA. USENIX Association.

[Ben-yehuda et al., 2006] Ben-yehuda, M., Mason, J., Krieger, O., Xenidis, J., Doorn, L. V., Mallick, A., y Wahlig, E. (2006). Utilizing iommus for virtualization in linux and xen. En *In Proceedings of the Linux Symposium*.

[Bu y Chandy, 2004] Bu, L. y Chandy, J. A. (2004). Fpga based network intrusion detection using content addressable memories. En *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '04, páginas 316–317, Washington, DC, USA. IEEE Computer Society.

[Che et al., 2010] Che, J., Yu, Y., Shi, C., y Lin, W. (2010). A Synthetical Performance Evaluation of OpenVZ, Xen and KVM. En *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, páginas 587–594.

[Dharmapurikar et al., 2003] Dharmapurikar, S., Krishnamurthy, P., Sproull, T., y Lockwood, J. (2003). Deep packet inspection using parallel bloom filters. En *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, páginas 44–51.

[Dumitrescu, 2008] Dumitrescu, C. F. (2008). Design patterns for packet processing applications on multi-core intel architecture processors.

[Edward Whalen, 2013] Edward Whalen (2013). How to Configure x86 Memory Performance for Large Databases Using Linux Huge Pages.

[ETSI, 2013] ETSI (2013). Network functions virtualisation (nfv), architectural framework. nfv 002 v1.1.1.

[EZChip, 2014] EZChip (2014). Np 5, 240gbps npu for carrier ethernet applications.

[Ganegedara y Prasanna, 2013] Ganegedara, T. y Prasanna, V. (2013). A high-performance ipv6 lookup engine on fpga. En *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, páginas 1–4.

[Garzarella et al., 2015] Garzarella, S., Lettieri, G., y Rizzo, L. (2015). Virtual device passthrough for high speed vm networking. En *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, páginas 99–110, Washington, DC, USA. IEEE Computer Society.

[Gibb et al., 2008] Gibb, G., Lockwood, J., Naous, J., Hartke, P., y McKeown, N. (2008). NetFPGA: An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers. *IEEE Transactions on Education*, 51(3):364–369.

[Habib, 2008] Habib, I. (2008). Virtualization with KVM. *Linux J.*, 2008(166).

[Honjo et al., 2010] Honjo, M., Kubota, A., y Kitamura, T. (2010). Parallel programming framework for heterogeneous computing environment with Xen virtualization. En *TENCON 2010 - 2010 IEEE Region 10 Conference*, páginas 1100–1105.

[Jiang y Prasanna, 2013] Jiang, W. y Prasanna, V. (2013). Data structure optimization for power- efficient ip lookup architectures. *Computers, IEEE Transactions on*, 62(11):2169–2182.

[Kachris et al., 2014] Kachris, C., Sirakoulis, G., y Soudris, D. (2014). Network function virtualization based on fpgas: A framework for all-programmable network devices.

[Kachris y Vassiliadis, 2006] Kachris, C. y Vassiliadis, S. (2006). Design of a web switch in a reconfigurable platform. En *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, páginas 31–40, New York, NY, USA. ACM.

[L. Wirbel, 2014] L. Wirbel (2014). Xilinx SDNet: A New Way to Specify Network Hardware.

[Lockwood et al., 2004] Lockwood, J. W., Moscola, J., Reddick, D., Kulig, M., y Brooks, T. (2004). Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection. En Wakamiya, N., Solarski, M., y Sterbenz, J., editors, *Active Networks*, number 2982 in Lecture Notes in Computer Science, páginas 44–57. Springer Berlin Heidelberg.

[M. Tim Jones, 2009] M. Tim Jones (2009). Anatomy of a Linux hypervisor.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., y Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

[Mitra et al., 2007] Mitra, A., Najjar, W., y Bhuyan, L. (2007). Compiling pcre to fpga for accelerating snort ids. En *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, páginas 127–136, New York, NY, USA. ACM.

[Moreno et al., 2014] Moreno, V., Santiago del Rio, P., Ramos, J., Garcia-Dorado, J., Gonzalez, I., Gomez Arribas, F., y Aracil, J. (2014). Packet Storage at Multi-gigabit Rates Using Off-the-Shelf Systems. En *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, páginas 486–489.

[Nobach y Hausheer, 2015] Nobach, L. y Hausheer, D. (2015). Open, elastic provisioning of hardware acceleration in nfv environments. En *Networked Systems (NetSys), 2015 International Conference and Workshops on*, páginas 1–5.

[Rizzo, 2012] Rizzo, L. (2012). Netmap: A novel framework for fast packet i/o. En *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, páginas 9–9, Berkeley, CA, USA. USENIX Association.

[Rizzo et al., 2013] Rizzo, L., Lettieri, G., y Maffione, V. (2013). Speeding up packet i/o in virtual machines. En *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, páginas 47–58, Piscataway, NJ, USA. IEEE Press.

[Shanley y Anderson, 1995] Shanley, T. y Anderson, D. (1995). *PCI System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[SIG, 2014] SIG, P. (2014). Pci express base specification revision 3.1.

[Sourdis et al., 2007] Sourdis, I., Pnevmatikatos, D. N., y Vassiliadis, S. (2007). Scalable multi-gigabit pattern matching for packet inspection.

[Vu et al., 2014] Vu, D. V., Sander, O., Sandmann, T., Baehr, S., Heidelberger, J., y Becker, J. (2014). Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using PCI express single-root I/O virtualization. En *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, páginas 1–6.

[Walters et al., 2014] Walters, J., Younge, A., Kang, D. I., Yao, K. T., Kang, M., Crago, S., y Fox, G. (2014). GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. En *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, páginas 636–643.

[Wicaksana y Sasongko, 2011] Wicaksana, A. y Sasongko, A. (2011). Fast and reconfigurable packet classification engine in fpga-based firewall. En *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, páginas 1–6.

[Xilinx, 2015] Xilinx (2015). Virtex-7 fpga gen3 integrated block for pci express v4.0 logicore ip product guide 023, vivado design suite.

[Yang y Prasanna, 2013] Yang, Y.-H. y Prasanna, V. (2013). Robust and scalable string pattern matching for deep packet inspection on multicore processors. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11):2283–2292.

[Zazo et al., 2014] Zazo, J., Forconesi, M., Lopez-Buedo, S., Sutter, G., y Aracil, J. (2014). TNT10g: A high-accuracy 10 GbE traffic player and recorder for multi-Terabyte traces. En *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, páginas 1–6.

[Zazo et al., 2015] Zazo, J., Martín, R., Leira, R., González, I., Sutter, G., y J., G. F. (2015). Clasificación de tráfico de red mediante aceleradores hardware. En *Jornadas de Computación Reconfigurable y Aplicaciones*.

A Tecnologías subyacentes

A.1 Virtualización

Es un hecho que la industria tecnológica se ha volcado considerablemente en la virtualización durante los últimos años. Sin embargo, el concepto no es nuevo y su creación se remonta a tiempo atrás. Esta sección provee una vista general a alto nivel de las distintas tecnologías de virtualización y los métodos más relevantes en la actualidad.

El concepto de virtualización se origina a finales de los años 60 y principios de los 70, donde predominaban los mainframes en el mundo de alta computación e IBM invertía un alto esfuerzo en el intento de crear soluciones compartidas en tiempo real. Con aplicaciones compartidas se entiende el uso de unos mismos recursos físicos por parte de un grupo de usuarios con el primordial objetivo llegar a un compromiso entre la eficiencia y el coste del computador. Este nuevo paradigma supone una brecha con las tendencias de la época: la posibilidad de manejar un ordenador sin poseerlo en realidad.

En la actualidad las razones para aplicar virtualización varían considerablemente pero una premisa está siempre presente: la capacidad de una estación de trabajo es tan grande que es muy difícil hacer un aprovechamiento eficiente de la misma. Una manera de explotar estos recursos es mediante la virtualización pero ésta, a su vez, permite simplificar el procesamiento de datos (estrategia *divide and conquer*) o la migración del sistema en caso de fallos o anomalías en la estación principal.

Los centros de procesamiento de datos (CPDs) se benefician de la abstracción de la capa hardware que supone emplear la virtualización. Agregar elementos que integren nuevas CPUs, memorias, discos o interfaces de red es un proceso ágil y escalable.

A.1.1 Clasificación de VMMs

Si se entiende la virtualización como la posibilidad de ejecutar distintos sistemas en un único anfitrión de manera concurrente, entonces es esencial la figura del hipervisor. El hipervisor, también llamado monitor de máquinas virtuales (VMM), crea una plataforma sobre la cual se ejecutan las distintas VMs. Es el hipervisor el que toma control completo de la plataforma hardware y, concurrentemente, atiende las peticiones de entrada y salida de datos por parte del software anfitrión. En otras palabras, el monitor tiene la competencia de aislar y controlar los recursos que pueden ser accedidos desde la VM.

La atención de las peticiones de entrada/salida (IO) se gestiona mediante el uso de dos técnicas ampliamente extendidas: emulación de dispositivos o virtualización, que deben ser evaluadas para sistemas donde el rendimiento o la flexibilidad seas componentes críticas.

- La emulación de dispositivos mediante software (CPU incluida), es la opción más flexible y portable. El código para la arquitectura invitada (MIPS, ARM, PPC, x86, etc.) es transformado en operaciones válidas para la arquitectura real del equipo. Se produce una reconversión de las instrucciones originales que, a todos los efectos, es un proceso demandante de una alta carga computacional y una degradación en el rendimiento general del sistema virtualizado es palpable. Este es el paradigma de herramientas libres como Qemu [Bellard, 2005] y aunque se haya hablado de instrucciones a nivel de procesador, las mismas características se aplicarían a cualquier otro dispositivo emulado, como la memoria secundaria o las interfaces de red. Mediante software debe adecuarse la entrada/salida de datos del componente ficticio al componente real. Por ejemplo, este es el caso de la emulación de una tarjeta de red de 100 Mbps. Aunque el sistema real pudiera contar con interfaces de tasa mayor, de ningún modo debería observarse tal efecto de cara al sistema operativo invitado.
- De manera complementaria, la virtualización se localiza en entornos donde tanto el sistema emulado como el físico presentan exactamente la misma arquitectura. Es el caso convencional de correr un sistema x86 sobre una máquina de las mismas características. Aunque el código no se puede ejecutar de manera nativa, dado que hay que analizar las instrucciones ejecutadas, tener un control de los permisos o realizar cambios en los direccionamientos a memoria, gran parte del código no tendrá que verse modificado en absoluto. A pesar de perder generalidad a la hora de portar el sistema (se liga el entorno a una arquitectura particular), las instrucciones no son alteradas y el rendimiento será más eficiente.

No obstante, no debe olvidarse que ambos paradigmas pueden ser combinados. Es el caso de un sistema virtual donde se simulen determinados periféricos pero se decida emplear el mismo juego de instrucciones que en el equipo anfitrión para no necesitar de una traducción del código al lenguaje nativo. Cuando Qemu es utilizado en conjunción con una KVM [Habib, 2008], Qemu se encarga de realizar las tareas de emulación de dispositivos, mientras que KVM permitirá la ejecución de instrucciones sobre la CPU física.

Si se pretende clasificar el hipervisor, dado que se trata del programa esencial en todo el proceso de virtualización, existen dos tipos según IBM [M. Tim Jones, 2009] prestando atención a su arquitectura y que reciben los nombres de tipo 1 y de tipo 2.

- Un hipervisor de tipo 1 (Figura A.2) se ejecuta directamente sobre el hardware del siste-

ma. En ocasiones estos VMMs son referidos como nativos, embebidos o *bare metal* en la literatura sobre el tema.

- Un hipervisor de tipo 2 (Figura A.1), por el contrario, requiere de un sistema operativo sobre el que ejecutarse. Cuando la virtualización despegó, esta alternativa se hizo extremadamente popular, ya que permitía a un administrador instalar y configurar un software sobre una estación previamente gestionada. Es el caso de herramientas como VMware WorkStation. El hipervisor se puede ejecutar tanto a nivel de usuario como sobre el núcleo sistema operativo (diagramas representados en la Figura A.1 respectivamente).

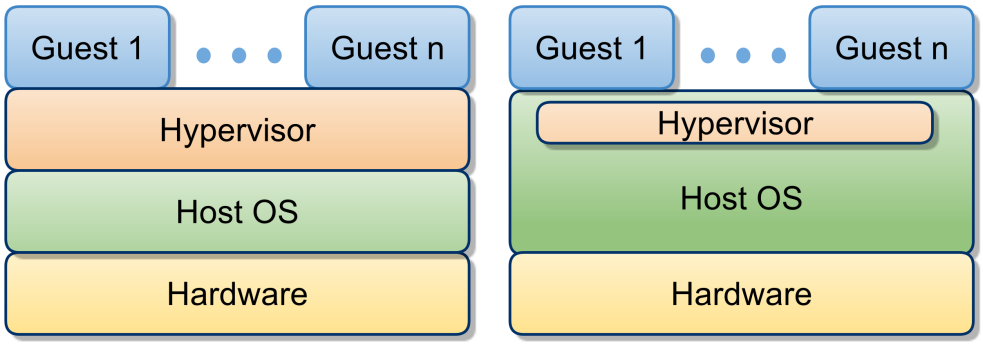


Figura A.1: Hipervisor de tipo 2.

Dependiendo del autor, el tipo 1 podría subdividirse en tipo 1 y tipo 0 (Figura A.3). De acuerdo con esta separación más granular, ambos hipervisores correrían sobre el hardware directamente. La particularidad reside en la necesidad por parte de los hipervisores del tipo 1 de apoyarse en un sistema operativo (conocedor del estado de virtualización). Sería éste quien proveyera de las pilas para operaciones de IO o controladores del hardware. El tipo 0 quedaría relegado a aquel entorno que no precisa de un sistema operativo embebido que facilite la virtualización. Es un VMM capaz de ejecutarse sobre un entorno sin *host*. Difiere del tipo 1 en tanto que el hipervisor no se integra en un sistema operativo que actúe de mediador. Hipervisores del tipo 1 es, por ejemplo, Xen. Casos particulares del tipo 0, podría ser VMware ESXi. Sin embargo, considerando estrictamente la ideología de este tipo podría no cumplirse realmente. A pesar de no necesitar de un sistema operativo anfitrión, la herramienta es acompañada por un kernel monolítico que permite la comunicación con el hardware y no está totalmente integrada en la capa del hipervisor (corre sobre ella).

A.1.2 Arquitectura de VMMs

Un VMM, a pesar de los tipos explicados, es exclusivamente una capa que abstrae al sistema virtualizado del hardware de la máquina anfitriona. De esta manera, cada invitado accede

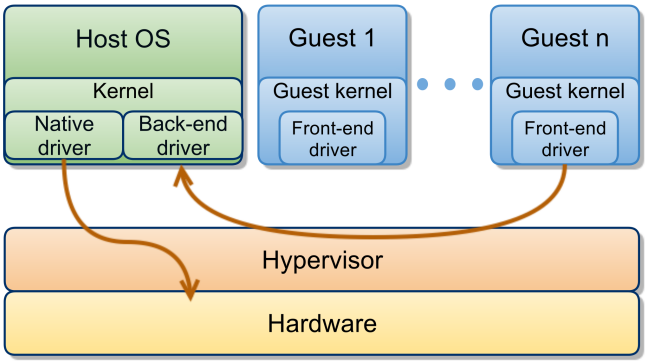


Figura A.2: Hipervisor de tipo 1.

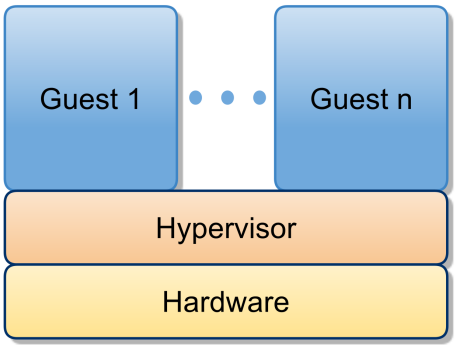


Figura A.3: Hipervisor de tipo 0.

a unos recursos virtuales en sustitución del hardware real. A continuación se enumeran los elementos genéricos internos a cualquier implementación de un hipervisor.

A un alto nivel, un VMM requiere de unos módulos mínimos para iniciar un sistema operativo anfitrión: un núcleo para arrancar (ya sea implementado por el propio hipervisor en el caso del tipo 0, o dependiente de otro sistema para otros tipos como el 1 y el 2), memoria persistente y uno o varios dispositivos de red. La memoria persistente y el dispositivo de red están, típicamente, asociados con los elementos físicos del equipo: disco físico de la máquina y tarjeta de red existente.

Finalmente, existen una serie de aplicaciones adicionales que deben estar presentes para poder ejecutar múltiples invitados simultáneamente. De manera similar a las llamadas que un programa realiza a las funcionalidades del kernel, una capa que controle las invocaciones de las funciones ejecutadas en modo kernel debe estar disponible. Esta capa es denominada como *hypercall layer*. Mientras tanto, periféricos de IO deben ser emulados en el kernel o asistidos por el código del sistema invitado. Las interrupciones deben ser capturadas exclusivamente por el hipervisor con el objeto de ser enrutadas hacia el invitado adecuado, de la misma forma que ocurre con el manejo de excepciones. Como ejemplo, póngase en la situación donde un error terminal obliga a bloquear un sistema. Éste debería congelar únicamente la VM que genera dicha interrupción mientras que para el resto debería ser invisible esta anomalía.

Los dos últimos elementos que debe aportar un VMM son: un enlazador de memoria (*mapper*) que asocie direcciones físicas de la memoria principal a un sistema particular y un planificador que seleccione a qué sistema se le asocia el quantum de tiempo actual. En la Figura A.4 quedan reflejados los aspectos básicos del VMM en una arquitectura del tipo 0 para un sistema con dos máquinas virtuales.

A modo de resumen, los elementos básicos son:

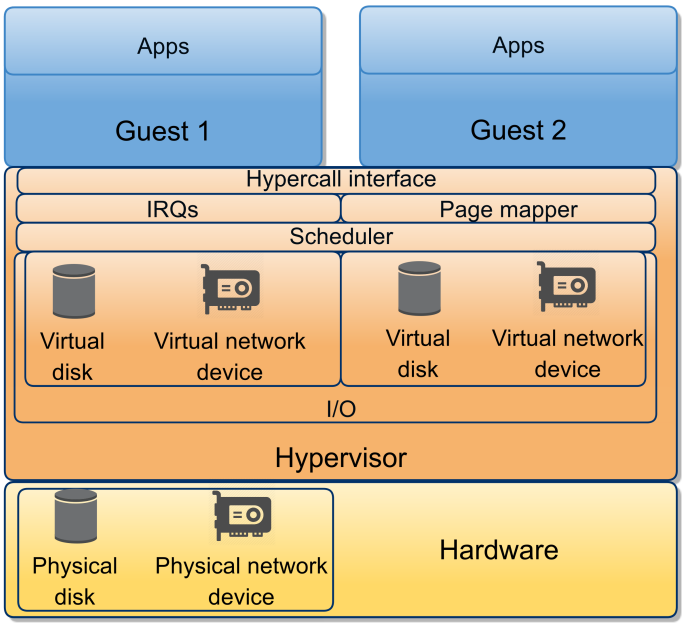


Figura A.4: Componentes de un hipervisor.

- Planificador de tareas, *scheduler*.
- *Mapper* de memoria física a direcciones virtuales para cada una de las máquinas virtuales.
- Control de las llamadas al kernel con el fin de asegurar el aislamiento y la seguridad entre los distintos invitados.
- Enrutado o manejo de las posibles interrupciones generadas, así como de los errores.
- Módulo de virtualización/emulación del hardware que permite abstraer al invitado del equipo real.
- Módulo de correspondencia entre hardware virtualizado y hardware real (memoria secundaria y dispositivos de red).

A.1.3 Herramientas de virtualización en la actualidad

Dos de las alternativas que se han impuesto durante los últimos años en el ámbito de la virtualización bajo entornos Linux son Xen [Honjo et al., 2010] y KVM [Habib, 2008] debido a la licencia libre de ambos proyectos. Xen es un hipervisor de tipo 1, es decir, se ejecuta directamente sobre el hardware pero con la ausencia de los mecanismos necesarios para ser capaz de manejarlo por si mismo plenamente. Depende de un sistema operativo que le permita administrar el sistema, denominado como dominio 0 por la aplicación y que soporte el uso de este hipervisor. Aunque pudiera pensarse que tener un VMM del tipo 1 es un atraso respecto del

Machine	Bespin	Delta
CPU (cores)	2xE5-2670 (16)	2xX5660 (12)
Clock Speed	2.6 GHz	2.6 GHz
RAM	48 GB	192 GB
Numa nodes	2	2
GPU	K20m	2x C2075

Tabla A.1: Equipos del rendimiento para distintos tipos de VMM [Walters et al., 2014].

tipo 0, un sistema con Xen emplea los drivers de un sistema operativo de amplia difusión, mejorando la compatibilidad con los distintos periféricos y la configuración de estos controladores puede ser realizada mediante las técnicas habituales (dependientes del sistema operativo).

En la otra mano, KVM se ejecuta sobre el propio kernel Linux, reutilizando gran parte de las herramientas presentes en el mismo. Es el caso del planificador de tareas que es reutilizado para la asignación de procesadores a las máquinas virtuales en el momento de ejecución. Usar software maduro, que ha sido estudiado y corregido a lo largo del tiempo generalmente aislará posibles errores e inestabilidades en el sistema.

En [Walters et al., 2014] se presenta el estudio de estas dos utilidades junto a otros paradigmas de virtualización diferentes: un hipervisor del tipo 0 comercial, VMware ESXi, y el concepto de contenedor de linux (LXC). LXC no ofrece plena virtualización en el sentido de que no es capaz de abstraerse del hardware del equipo. Su funcionalidad queda limitada a simular distintos entornos de usuario (*userspace*) y proveer de aislamiento para los distintos sistemas/procesos. La penalización en rendimiento por su uso es teóricamente menor.

El objeto de evaluación del trabajo mencionado es un sistema de altas prestaciones, donde se conecta una GPU directamente a una VM. El paradigma de desarrollo para GPUs involucra tanto la evaluación de CUDA como de OpenCL. Los ordenadores de referencia quedan descritos en la Tabla A.1 y el principal motivo de recopilar la información de este trabajo es observar la tendencia evolutiva de los hipervisores ante el aumento de los recursos del hardware subyacente.

El sistema anfitrión posee un kernel de Linux 3.12 para las pruebas con Xen y KVM. Para el caso de LXC se trata de la versión 2.6.32-358.23.2. Cada uno de los sistemas virtualizados es un CentOS 6.4 con el kernel de fábrica (2.6.32-358.23.2), CUDA versión 5.5 , 20 GB de RAM asignada y 6 u 8 cores (la totalidad de cores disponibles en uno de los procesadores). Se respeta la correcta asignación de los nodos núcleos para que se correspondan con un mismo nodo NUMA, al igual que se tiene en consideración que la GPU no se vea penalizada por esta clase de arquitectura.

Los resultados obtenidos al analizar los algoritmos de LAMMPS (simulador molecular), GPU-LIBSVM (aprendizaje automático basado en máquinas de vector de soporte) y LULESH (simulador para aplicaciones hidrodinámicas) sobre una GPU señalan la diferencia en rendimiento en función de la herramienta. KVM resulta favorecida en estos experimentos hasta el punto de que se llegan a alcanzar desviaciones superiores al 10 %, especialmente en el caso del hipervisor Xen (seguido de VMware ESXi). Estas diferencias no reflejan fielmente la diferencia que debería haber en función del paradigma del hipervisor, más bien, apuntan a la correcta optimización de cada una de las implementaciones evaluadas. LXC ofrece un rendimiento cercano al nativo pero con la particularidad de que únicamente se permite virtualizar sistemas Linux. Esto favorecerá que el desarrollo de este trabajo se centre en un entorno sostenido en el VMM KVM dado que las tecnologías usadas para virtualizar una tarjeta de red, o un dispositivo hardware reconfigurable, serán en gran medida análogas a las empleadas por una GPU.

No obstante, no debe obviarse que no existe una arquitectura que resulte superior a las otras en todos los casos [Che et al., 2010]. En este artículo previo al resultado con GPUs se evalúan distintos tests para medir el comportamiento general. Los bancos de pruebas empleados para la evaluación son: SPEC CPU2006, LINPACK, compilación del Kernel de Linux, RAMSPEED, LMbench, IOzone, Bonnie++, NetIO, WebBench, SysBench y SPEC JBB2005, donde la virtualización basada en contenedores ofrece mayor rendimiento sobre las alternativas Xen y KVM. Siendo este segundo ligeramente inferior en los mencionados escenarios. Es necesario, por tanto, buscar un compromiso entre portabilidad (facilidad de uso) y rendimiento. De acuerdo a ambos artículos, la virtualización del procesador no es el principal cuello de botella, que se localiza en los fallos en el acceso a memoria, manejo de interrupciones e IO.

Al mismo tiempo, se pone de manifiesto que los procesadores con arquitectura actuales mejoran la virtualización, ofreciendo un mayor soporte desde la capa hardware. Se desmiente, en cierto sentido, la creencia tradicional donde se confronta la viabilidad de virtualizar en entornos de rendimiento crítico. Esta idea se ha quedado desfasada.

A.2 *PCI passthrough*

Compartir un dispositivo entre distintas máquinas virtuales conlleva una penalización no despreciable que puede acentuarse en función del paradigma de virtualización empleado. Si la emulación del dispositivo se realiza en el nivel del hipervisor (o en el espacio de usuario de una VM) este hecho se acentúa. Sin embargo, este consumo adicional sólo tiene relevancia mientras que se necesite compartir un dispositivo entre los diferentes sistemas invitados. En el momento en el que no se precise de esta repartición, existen métodos más convenientes para el acceso a estos dispositivos.

Visualizando el problema desde un alto nivel, aplicar *PCI passthrough* a un dispositivo consiste en el aislamiento del mismo para que su accesibilidad se restrinja a un único invitado (ver Figura A.5). Aunque la necesidad de ligar un dispositivo exclusivamente a una VM no parezca significativamente ventajoso inicialmente, existen determinados escenarios donde realmente merece la pena. Dos son los escenarios primordiales: asegurar el rendimiento crítico del dispositivo y proveer de acceso exclusivo al dispositivo. Éste es el caso donde por la política de acceso se prefiere que su compartición no esté habilitada (por ejemplo, un adaptador de vídeo).

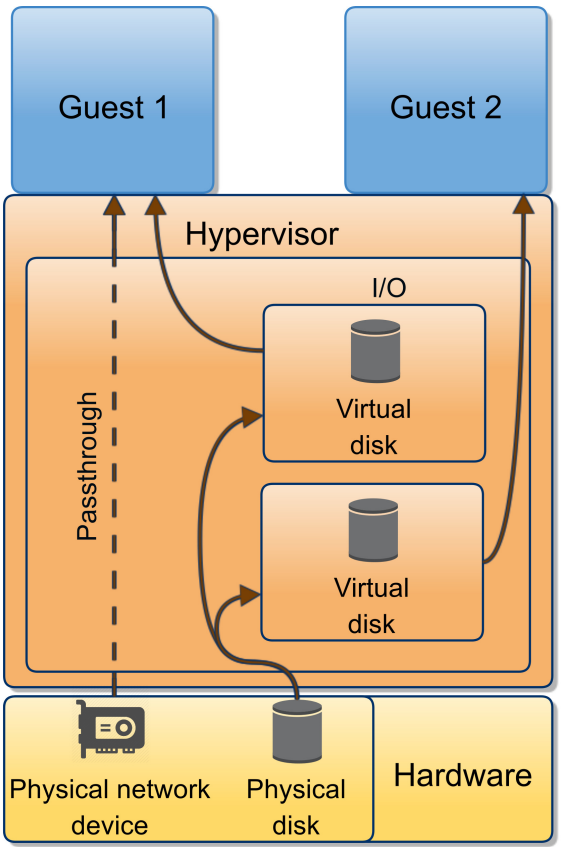


Figura A.5: Tecnología *PCI passthrough*.

Referido al apartado de rendimiento, unas prestaciones similares a las nativas son logradas con esta técnica. La convierten en una metodología perfecta para aplicaciones de red debido a la alta carga de IO asociada y que en gran medida se veían hasta el momento limitadas por el VMM.

Tanto Intel como AMD proveen del soporte hardware necesario para realizar *PCI passthrough* en sus procesadores actuales (en conjunción con nuevas instrucciones que asisten al VMM). Intel se refiere a estas opciones como Virtualization Technology for Directed IO (VT-d) mientras que AMD prefiere utilizar el término de IO Memory Management Unit (IOMMU). En ambos casos la CPU es capaz de ligar direcciones físicas del dispositivo PCI a direcciones virtuales en el entorno del sistema invitado. Cuando esta asociación ocurre, el hardware controla

el acceso (y ofrece protección) mientras que el sistema operativo utiliza el periférico como si se tratara de un elemento no virtualizado. A la par que se enlaza un invitado a dichas direcciones físicas, el aislamiento del elemento hardware es provisto con el fin de que ningún otro invitado (o el anfitrión) puedan acceder a él.

El principal problema que ocurre con la tecnología *PCI passtrough* usada de modo independiente es la rigidez de la solución cuando se requiere migrar los servicios de estación. La migración, suspensión de la máquina virtual y posterior configuración en un nuevo sistema anfitrión, puede afectar en la misma medida que la configuración del hardware haya variado. Es decir, si las direcciones físicas nunca más se corresponden con las originales se deberá reiniciar el sistema virtualizado para adoptar la configuración idónea. Adicionalmente a este problema, esta el profundo aislamiento que del dispositivo se produce. Mientras éste se encuentre en uso por uno de los invitados, ningún otro será capaz de aprovecharlo para su uso incluso en el caso en el que se encontrase ocioso.

A.3 SR-IOV

El siguiente paso en virtualización de periféricos es una tecnología completamente novedosa a día de hoy. El concepto clave en este proceso es denominado como SR-IOV. Esta tecnología (desarrollada por el PCI-Special Interest Group) permite a un único dispositivo figurar como múltiples. Con SR-IOV el dispositivo hardware puede exportar no sólo PFs sino también VFs. Estos dos conceptos son clave en virtualización de dispositivos PCIe y presentan ligeras diferencias.

- Una PF representa un dispositivo PCIe plenamente funcional con soporte para SR-IOV en su espacio de configuración. Presentan un espacio propio de configuración y son capaces de manejar (creación y destrucción) funciones virtuales.
- En el otro extremo, una VF es un dispositivo PCIe reducido con soporte únicamente para procesar IO. Cada función virtual está asociada a una PF y el número máximo está limitado por el propio periférico (o el límite superior de 255 VF por cada PF).

El hipervisor puede realizar la asociación de una, o más, funciones virtuales a una VM. Este ligado excluye al resto de VMs de la posibilidad de acceder al recurso. Sin embargo, al poder crear diversas VFs la limitación impuesta por el uso de *PCI passthrough* se ve solventada. El rendimiento de las funciones virtuales es cercano al nativo y ofrece un mejor rendimiento que la paravirtualización y emulación de dispositivos.

Para profundizar en la implementación de SR-IOV es necesario unas nociones previas de PCI. Un dispositivo PCI se define con la notación *Bus Device Function (BDF)*. En el uso más

sencillo, el bus hace referencia al puerto de comunicaciones PCI, la ubicación física se corresponde con el número del dispositivo y el número de función permitirá distinguir entre las distintas características ofertadas por el hardware. De este modo, el sistema operativo al iniciarse enumera los slots de manera geográfica e intenta leer el vendedor del dispositivo para la función 0 (de obligada implementación). Si el valor devuelto es 0xFFFFFFFF, vendedor inválido, la combinación BDF no está activa y no hay un dispositivo físico durante la ejecución. Si la función 0 no está presente, no es necesario seguir consultando sobre el resto de posibles funciones dado que tampoco existirán. Si se detecta un dispositivo válido la BIOS (o el sistema operativo) se encargan de realizar el mapeo de memoria y la asignación de las direcciones de los puertos de IO. Estos valores permanecerán válidos tanto tiempo como el sistema operativo siga funcionando. Este proceso se aplica de manera recursiva sobre cada uno de los posibles buses y dispositivos del sistema.

31 16	15 0	
Device ID	Vendor ID	DWORD 0
Status	Command	DWORD 1
Class code	Rev ID	DWORD 2
BIST / Header typ	Lat.timer / Cache lin	DWORD 3
BAR0		DWORD 4
BAR1		DWORD 5
BAR2		DWORD 6
BAR3		DWORD 7
BAR4		DWORD 8
BAR5		DWORD 9
Cardbus CIS pointer		DWORD 10
Subsystem ID	Subsystem vendor ID	DWORD 11
Expansion ROM Base Address		DWORD 12
Reserved	Cap Point	DWORD 13
Reserved		DWORD 14
Max Lat / Min Gnt / IRQ pin	IRQ line	DWORD 15

Cuadro A.1: Espacio de configuración de un dispositivo PCIe.

El mecanismo por el que un dispositivo genérico informa al sistema operativo de las capacidades que ofrece está estandarizado y se conoce como espacio de configuración (Cuadro A.1). Todos los dispositivos PCI (excepto los bus bridges) deben proveer un registro de 256 bytes para dicho propósito y es ampliado a la cantidad de 4096 bytes para dispositivos PCIe.

31 24	23 20	19 16	15 0	
Next cap	Ver		Cap Id	DWORD 0
SR–IOV capabilities				DWORD 1
SR–IOV status			SR–IOV control	DWORD 2
Total VFs			Initial VFs	DWORD 3
RsvdP	Function dependency link		Num VFs	DWORD 4
VF stride			First VF offset	DWORD 5
VF device id			RsvdP	DWORD 6
Supported page sized				DWORD 7
System page size				DWORD 8
VF BAR0				DWORD 9
VF BAR1				DWORD 10
VF BAR2				DWORD 11
VF BAR3				DWORD 12
VF BAR4				DWORD 13
VF BAR5				DWORD 14
VF migration state array offset				DWORD 15

Cuadro A.2: Capacidad SR-IOV en un dispositivo PCIe.

Información de propósito general es facilitada mediante este medio tal como el identificador del dispositivo y fabricante o las direcciones base de cada uno de los BAR. Información más concreta es capaz de obtenerse en los dispositivos a partir de la dirección dada por el puntero *capabilities pointer*. En esta configuración se localiza un identificador de la capacidad (definidos a priori) y un puntero a la siguiente estructura. De manera análoga existe un un puntero a las capacidades extendidas que queda limitado a PCIe. La estructura de configuración para SR-IOV queda contemplada en el Cuadro A.2. Los registros que debe proveer un diseño hardware que presente la tecnología SR-IOV son, por tanto:

- SR-IOV *capabilities*. Entre las competencias que debe especificarse está la posibilidad de migración de VF y el número de la interrupción usada para las operaciones de migración. Sin embargo, la posibilidad de migración de VFs está más allá de la capacidades de SR-IOV (su uso está ideado para sistemas que implementen virtualización multinodo). Por ende, el valor de este registro debería ser 0x00 para la virtualización de un único nodo.
- SR-IOV control.

  0 *VF enable*. Indica si las VFs están habilitadas/deshabilitadas.

1 *VF migration enable*. Indica si la migración está permitida (0 para SR-IOV).

2 *VF migration interrupt enable*. Indica si las migraciones deben ser notificadas mediante el uso de interrupciones.

3 *VF memory space enable*. Indica si las VFs tienen su propio espacio de memoria.

4 *VF ARI enable*. Habilitado por software si el direccionamiento por ARI está habilitado por los recursos hardware en los que está sustentado el dispositivo. ARI ofrece una interpretación alternativa del ID del dispositivo. En lugar de la notación BDF, función y dispositivo son utilizados conjuntamente para ofrecer hasta un máximo de 255 funciones (se considera que el dispositivo es 0 y únicamente puede existir un dispositivo por bus). El PF puede utilizar este valor para calcular valores óptimos de *First VF offset* y *VF stride*.

15:5 Reservado.

- SR-IOV *status*.

  0 *VF migration interrupt pending*. Indica si una migración presenta una interrupción pendiente de ser atendida. Sin uso en SR-IOV.

  15:1 Reservado.

- *Total VFs*. Número total de funciones virtuales que pueden llegar a ser asociadas a una PF particular.

- *Initial VFs*. Número máximo de funciones virtuales reservadas en este PF.

- *Num VFs*. Número actual de funciones virtuales en uso.

- *First VF Offset*. Offset del requester ID de la primera VF.

- *VF Stride*. Offset del requester ID entre las sucesivas VF. Por lo tanto, la fórmula para calcular el identificador del requester de una función virtual n es:

$$RID_{VF} = RID_{PF} + First\ VF\ Offset + (n-1) \times (VF\ Stride)$$

- *Function dependency link*. Contiene el número de función física de la propia PF si no existen dependencias (o es el último elemento en la lista de dependencias). En caso contrario, es una lista enlazada de PFs que deberían reservar sus VFs de modo coordinado. Por ejemplo, un dispositivo de red con una PF como interfaz de red y otra PF como módulo criptográfico. A pesar de estar definidas como funciones físicas distintas, la función criptográfica podría acelerar la capacidad de red (funciones virtuales dependientes entre si).

- *VF Device ID*. En el espacio de configuración de la función virtual, device y vendor deben valer 0xFFFF.

Nombre del campo	Bit	PF	VF
IO space enable	0	Base	0
Memory space enable	1	Base	0
Bus master enable	2	Base	Base
Parity error enable	6	Base	RsvdP
SERR enable	8	Base	RsvdP
Interrupt disable	10	Base	0

Tabla A.2: Diferencias en el *command register* entre PFs y VFs.

Nombre del campo	Bit	PF	VF
Interrupt status	3	Base	0

Tabla A.3: Diferencias en el *status register* entre PFs y VFs.

- Aspectos relacionados al tamaño de página: permiten al software alinear al tamaño de página cada VF BAR.

- *Migration State Array Offset*. No aplicable para el caso de SR-IOV pero en el caso general permiten conocer el estado de la migración de una función virtual.

Por tanto, cada PF y VF tendrán su propio espacio de configuración como quedó reflejado en el Cuadro A.1 pero, sin embargo, presentan diferencias notorias que se detallan en la Tabla A.4. Entre ellas, competencias para la configuración del dispositivo, orientando su funcionalidad hacia el tratamiento de IO

A modo de recapitulación, aunque la virtualización ha estado en el aire por más de 50 años, ahora se vive un momento de gran interés, especialmente en su vertiente asociada a los dispositivos de IO. Procesadores comerciales con soporte para virtualización apenas cuentan con 5 años. En esencia, exclusivamente unas pocas aplicaciones han aparecido en este entorno y en un futuro cercano dado el gran impacto de arquitecturas como las asociadas a la computación en la nube, esta clase de tecnologías se tornarán en conceptos críticos durante los próximos años.

Sin embargo, al igual que ya se viera con *PCI passthrough*, la migración en tiempo real es únicamente experimental. Se requieren de configuraciones idénticas para que dicha translación pueda realizarse sin inconvenientes.

Nombre del campo	PF	VF
Vendor ID	Base	0xFFFF
Device ID	Base	0xFFFF
Command register	Base	Ver Tabla A.2
Status register	Base	Ver Tabla A.3
Class code	Base	Mismo valor en cada PF:VF
Revision ID	Base	Mismo valor en cada PF:VF
Cacheline size	Base	0x00
Latency timer	Base	0x00
Header type	Base	0x00
BIST	Base	0x00
Base Address Registers	Base	Valores propios
Cardbus CIS Pointer	Base	0x00
Subsystem Vendor ID	Base	Mismo valor en cada PF:VF
Subsystem Device ID	Base	Mismo valor en cada PF:VF
Expansion ROM BAR	Base	Valores propios
Capabilities pointer	Base	Base
Interrupt line	Base	0x00
Interrupt Pin	Base	0x00
Minimum GNT	Base	0x00
Maximum Latency	Base	0x00

Tabla A.4: Diferencias en el espacio de configuración entre PFs y VFs.

A.4 Interrupciones

A.4.1 Legacy

En los primeros instantes, una interrupción no era más que una señal que un dispositivo enviaba a una CPU, informando a esta última que el dispositivo requiere de su atención, indicando la necesidad de parar la actividad actual para responder al periférico. Esta acción puede postergarse en el tiempo si la rutina que se está ejecutando en ese instante de tiempo posee una prioridad mayor que la de la interrupción. En cualquier otro caso, la CPU detiene el proceso actual hasta que el manejador de interrupciones retorna, momento en el que se dispone del permiso para recuperar la tarea suspendida.

Pero esta alternativa pronto se tornó inviable, coincidiendo con que el número de periféricos se incrementaba. Se reemplaza la conexión directa de los elementos hardware al procesador por su conexión a un programmable interrupt controller (PIC). La tarea de multiplexación y

asignación de prioridades queda relegada al controlador. De este modo, un dispositivo que precisa de atención de la CPU se lo notifica al PIC en primera instancia. Si el registro IMR (Interrupt Mask Register) indica que la interrupción no debe ser procesada, ninguna acción adicional será llevada a cabo. En cualquier otro caso, el PIC alzará la señal de interrupción conectada a la CPU. Posteriormente, un doble reconocimiento por parte de la CPU es esperado a través de la señal INTA. En el primero, IRR (Interrupt Request Register) e ISR (In-Service Register) son actualizados. IRR indica qué interrupciones están pendientes de reconocimiento (doble asentimiento por parte de la CPU), mientras que ISR indica las interrupciones que están actualmente en ejecución (la CPU sigue en proceso de completar la rutina asociada con las mismas).

Con la segunda aserción de INTA, el par identificador y número son puestos en el bus del sistema. Con esta información la CPU puede localizar la entrada asociada en la tabla de interrupciones (IVT) y detectar la dirección sobre la rutina manejadora. A continuación se guardan las datos mínimos para realizar el cambio de contexto: banderas y punteros a la pila e instrucción actual. Una vez conocida la ubicación en memoria de la rutina de atención a la interrupción (ISR), se puede completar el proceso.

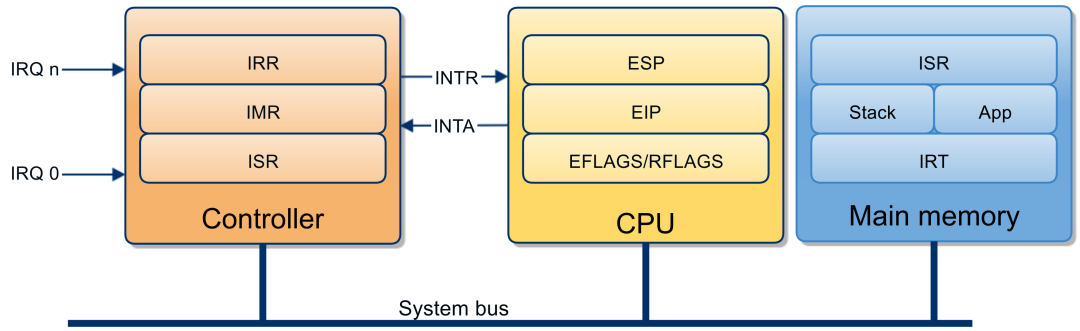


Figura A.6: Aproximación tradicional para la generación de interrupciones.

Este modelo de interrupciones presenta el inconveniente de ser lento al involucrar uno (o varios) controlador(es) que están limitados a un máximo de dos interrupciones encoladas (una por ISR y otra por IRR). Además, existe la contraindicación de que si una interrupción es compartida entre múltiples dispositivos, el sistema operativo se verá obligado a invocar a todas las rutinas de atención de interrupciones asociadas, derivando en una degradación del rendimiento global del sistema. Adicionalmente, los dispositivos PCI generalmente sólo pueden soportar una interrupción basada en pin o *legacy*, por lo que es frecuente que el driver posteriormente a la interrupción deba averiguar cuál es el evento que causó dicho fenómeno.

A.4.2 MSI

La invención que ayuda a aumentar el número de máquinas virtuales a la par que el rendimiento en la atención de generaciones es MSI [Shanley y Anderson, 1995]. En lugar de depender de conexiones físicas para generar interrupciones, MSI transforma estas interrupciones en mensajes. De este modo, la generación se sustenta en transmitir mensajes especiales encapsulados bajo PCI. MSI se presentó con la revisión 2.2 de PCI y ha sido conservado con las sucesivas revisiones de PCIe.

31 16	15 8	7 2	1 0	
Control Reg	Next Id	Cap Id = 0x05		DWORD 0
LSB Addr Reg			00	DWORD 1
Message Data Reg				DWORD 2

Cuadro A.3: Registro MSI de un dispositivo PCI para arquitecturas de 32b.

31 16	15 8	7 2	1 0	
Control Reg	Next Id	Cap Id = 0x05		DWORD 0
LSB Addr Reg			00	DWORD 1
MSB Addr Reg				DWORD 2
Message Data Reg				DWORD 3

Cuadro A.4: Registro MSI de un dispositivo PCI para arquitecturas de 64b.

Este cambio de filosofía es ideal para la virtualización de IO dado que de manera sencilla se logra el aislamiento de las distintas fuentes de datos frente a pines físicos que obligan a la multiplexación de las señales o al rutado mediante software. En la Figura A.6 se muestran los elementos involucrados en la generación de una interrupción por el método clásico y que permitirán apreciar las sustanciales mejoras que incorpora MSI.

Como punto de partida, cualquier dispositivo PCI que soporte interrupciones MSI debe mostrar entre sus competencias (*capabilities*) aquella con identificador 0x05. Los registros asociados a las competencias de MSI quedan contemplados en el Cuadro A.4 para arquitecturas de 64 bits y en el Cuadro A.3 para arquitecturas de 32 bits.

- El registro de control indica las condiciones actuales que presenta el dispositivo hardware:

  0 *MSI enable*. Este bit indica si MSI está habilitado (bit a 1) o deshabilitado (bit a 0).

  3:1 *Multiple message capable*. El sistema puede determinar a partir de este valor el número total de mensajes que el periférico estaría dispuesto a reservar. Siempre es poten-

cia de 2 y el máximo valor posible es 101b (32 interrupciones).

6:4 *Multiple message enable*. Después de que el software conozca el número máximo total de mensajes que el hardware soporta podrá habilitar hasta un número igual que el indicado en el valor de multiple message capable. Con cada nueva entrada configurada se incrementará el presente registro como si de un contador se tratara.

7 *Bit address capable*. Valor booleano indicando si los mensajes de interrupción generan direcciones de memoria de 64 bits. Un valor a 0 indica que las direcciones son de 32 bits.

15:8 Sin uso.

- El registro de direcciones no debe ser interpretado como una única dirección física y entre sus campos se presenta:

1:0 Los dos bits menos significativos son 0 (alineación en memoria a palabra de 4 bytes).

2 Bit de *Destination Mode (DM)*. Si RH=1 y DM=0 el campo ID del destinatario está en modo de direccionamiento físico. Es decir, únicamente el procesador del sistema que coincida con el mismo APIC ID es considerado para atender la interrupción. Si RH=1 y DM=1, el campo ID del destinatario es interpretado en modo lógico y la redirección está limitada sólo a aquellos procesadores que forman parte del grupo de procesadores lógicos basándose en APIC ID y el campo ID del destinatario.

3 Bit de *Redirection Hint (RH)*. Si RH=0 la interrupción es dirigida al procesador listado en el campo ID del destinatario. Si por el contrario RH=1, la interrupción es dirigida al procesador con menor prioridad de la lista de procesadores.

19:12 ID del destinatario.

63:20 Dirección base alineada a un bloque de memoria de 1MB.

- En última instancia, el registro de dato MSI presenta el siguiente formato:

7:0 Valor entero que indica el número de interrupción asociado con el mensaje.

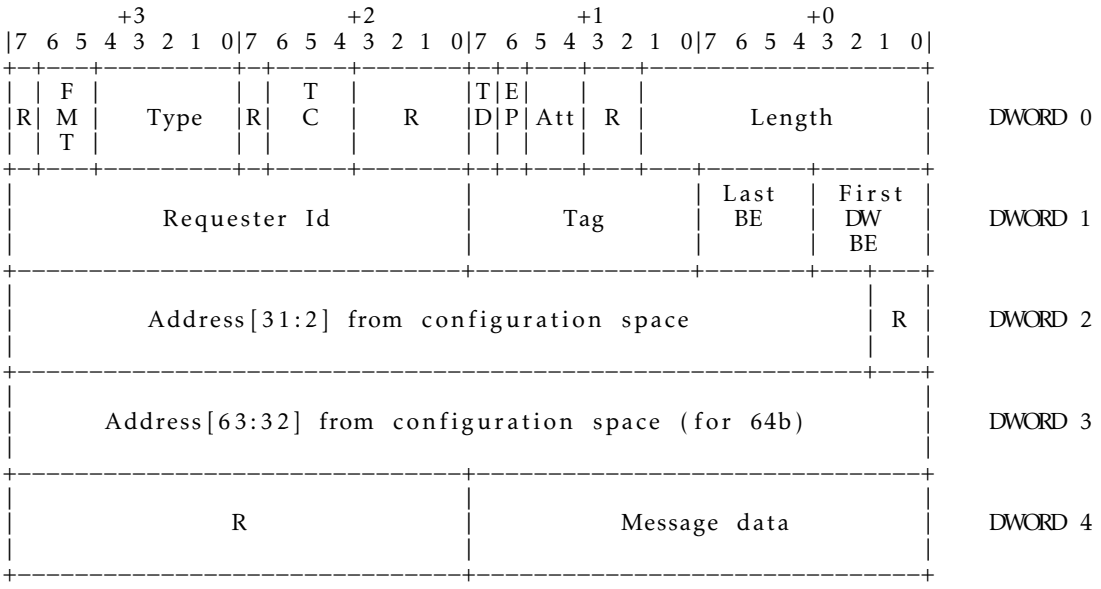
11:8 Modo de envío. Indica como la interrupción es manejada: Menor prioridad, SMI, SNI, etc. [Abbott, 2000].

13:12 Reservado.

14 *Trigger level*. Valor booleano indicando si la interrupción debe ser asentida o no.

15 *Trigger mode*. 0 para modo edge, 1 para modo level.

Un dispositivo con MSI habilitado interrumpirá a la CPU mediante la escritura en memoria de un paquete con un *payload* total de 3 dword (3×4 bytes). En el Cuadro A.5 se muestra la estructura de una interrupción MSI.

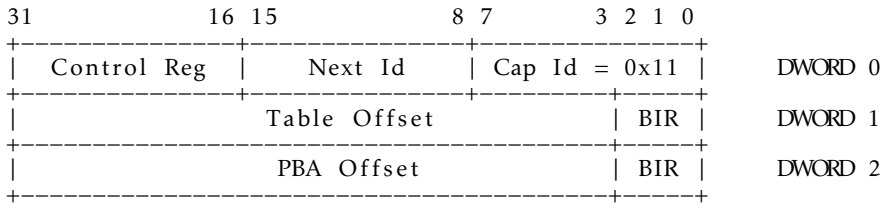


Cuadro A.5: Formato paquete de interrupción MSI.

Una escritura en memoria referida a la ubicación particular apuntada por la dirección del espacio de configuración de MSI para el dispositivo PCI y con una única palabra es lo que se conoce como vector MSI. Para determinar el número de la interrupción, como se ha visto previamente, se recurre al campo *message data*. La interpretación de los últimos bits del mismo como un valor entero indica el número de interrupción a generar.

A.4.3 MSI-X

El uso de una única dirección al que el usuario queda limitado en MSI se encontró restrictivo para algunos escenarios (como aquel en el que se pretende atender distintas interrupciones por distintos procesadores). En la especificación de PCI 3.0, se definió por primera vez MSI-X. Sustentándose en el concepto original de MSI, permite generar hasta un total de 2048 interrupciones. MSI-X permite generar un número mayor de interrupciones presentando cada una diferentes palabras de dirección y dato que simplifica la correspondencia entre interrupciones y procesador que debe atender la petición. El Cuadro A.6 muestra la entrada en el espacio de configuración asociado a MSI-X.



Cuadro A.6: Registro MSI-x en el espacio de configuración para un dispositivo PCIe.

- El registro de control incluye los siguientes campos:

  10:0 Tamaño de la tabla. Indica el número de entradas (menos 1) en la tabla de vectores. Así por ejemplo, el máximo valor representable es 0x7FF que equivale a 0x7FF+1, 2048 interrupciones.

  14:11 Reservado.

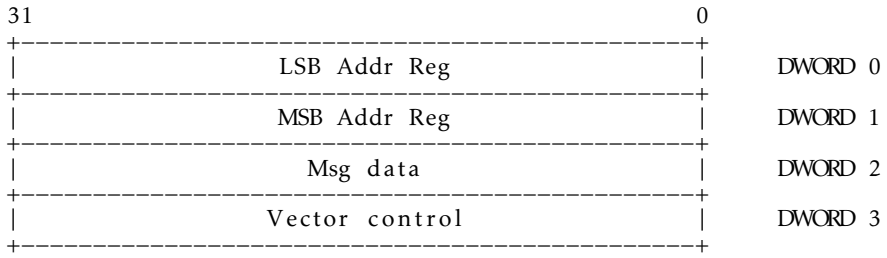
  15 MSI-X habilitado. Valor booleano indicando si las interrupciones MSI-X están soportadas por el dispositivo.

- Offset de la tabla en memoria del dispositivo hardware y registro indicador del BAR (BIR).

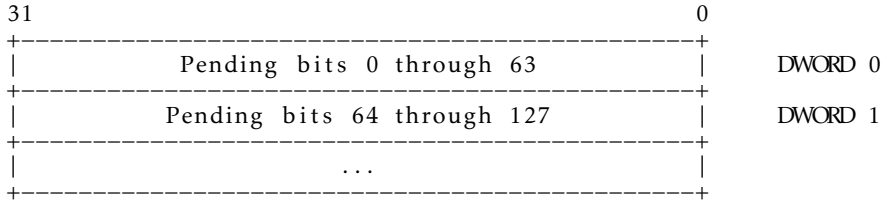
- Offset del pending bit array (PBA) en memoria del dispositivo hardware y registro BIR.

En contraste con la estructura de MSI, que directamente incluye todo la información de los vectores, MSI-X directamente incluye punteros a las estructuras. Éstas son mapeadas en memoria del dispositivo.

Cada vector constituye una entrada en la tabla MSI-X y típicamente contiene los campos mostrados en el Cuadro A.7. El PBA indica las interrupciones en proceso de ser atendidas a modo de máscara de bit. Su estructura queda contemplada en el Cuadro A.8.



Cuadro A.7: Entrada MSI-x en la tabla de vectores.



Cuadro A.8: Estructura Pending Bit Array (PBA).

- El campo de dirección es especificado por el software y debe estar siempre alineado a palabras de 4 bytes.

- El campo de dato preserva el formato que ya se describió para el caso de MSI.

- Del registro de control únicamente se emplea el bit 0. En el caso de valer 1, el sistema es incapaz de enviar un mensaje asociado a dicha interrupción. Es decir, queda completamente deshabilitada.

- El campo de bit pendiente se entiende como:

  63:0 Para cada bit asentido, existe un mensaje pendiente para la entrada MSI-X asociada. Aquellos bits que no tienen una entrada asociada no deben ser usados y deben ser establecidos a 0. Esta estructura es administrada única y exclusivamente por el hardware.

El formato de un mensaje es análogo al especificado en el Cuadro A.5. A modo de conclusión, tanto MSI como MSI-X son características del estándar de PCI que mejoran el manejo en los sistemas multiprocesador actuales mediante la reducción global de la carga de este proceso al disminuir la latencia asociada al tratamiento de una interrupción. Además, se ha hecho un importante esfuerzo en mejorar la escalabilidad de la tecnología, ideal para aplicaciones demandantes de una alta carga de entrada y salida de datos.

A.5 Formato de los datos en *endpoint* de PCIe

La descripción detallada de cada uno de los campos en función del tipo de paquete queda reflejada a continuación:

- Paquetes de petición generados en software. Cuadro 4.1 (notar que campos a 'R' (*reserved*) indican la carencia de sentido en ese tipo particular).

  1:0 *Address Type*:

  - 00 La dirección no ha sido traducida (es una dirección física asignada por el sistema).
  - 01 El paquete es una petición de traducción de dirección.
  - 10 La dirección ha sido previamente traducida (al espacio de direcciones usado en hardware).
  - 11 Reservada.

  63:2 *Address*. Dirección a la primera palabra referenciada por la petición. *FIRST_BE* debe ser usado para determinar los bytes válidos.

  74:64 *DWORD count*. Estos 11 bits indican el tamaño (en DWORDS) para ser escritos o leídos.

78:75 *Request type*. Identifica el tipo de operación (solamente a desarrollar tipos 0b0000 y 0b0001):

- 0000 *Memory read request*.
- 0001 *Memory write request*.
- 0010 *IO read request*.
- 0011 *IO write request*.
- 0100 *Memory fetch and add request*.
- 0101 *Memory unconditional swap request*.
- 0110 *Memory compare and swap request*.
- 0111 *Locked read request*. Sólo válido para dispositivos *legacy*.
- 1000 *Type 0 configuration read request*.
- 1001 *Type 1 configuration read request*.
- 1010 *Type 0 configuration write request*.
- 1011 *Type 1 configuration write request*.
- 1100 *Any meesage, except ATS and Vendor-Defined message*.
- 1101 *Vendor-Defined message*.
- 1110 *ATS message*.
- 1111 *Reserved*.

95:80 *Requester ID*. Identificación del solicitante. Por defecto, se utiliza la notación BDF pero si ARI está habilitado *device* y *function* son combinados.

103:96 *Tag*. Tag asociado a la petición. Cuando una petición es *non-posted* la lógica de usuario debe almacenar este valor y proveerlo en el paquete de respuesta.

111:104 *Target function*. Este campo indica el número de función a la que la petición está asociada.

114:112 *Bar ID*. Indica el BAR al que va dirigida la petición:

- 000 BAR 0, para VF VF-BAR 0.
- 001 BAR 1, para VF VF-BAR 1.
- 010 BAR 2, para VF VF-BAR 2.
- 011 BAR 3, para VF VF-BAR 3.
- 100 BAR 4, para VF VF-BAR 4.
- 101 BAR 5, para VF VF-BAR 5.

120:115 *Bar Aperture*. Este campo es empleado para determinar el tamaño total de la memoria. Por ejemplo, un valor de 12, indica que el BAR asociado tiene un tamaño total de 4 KB (2^{12} bytes) y, por ende, se pueden ignorar los bits [63:12] de la dirección.

123:121 *Transaction class (TC)*. Es un identificador usado para crear canales virtuales. Estos canales virtuales son simplemente conjuntos de datos lógicos separados con créditos y contadores distinguidos.

126:124 *Attributes*. Por defecto, al valor indicado por el *requester* y en el caso de peticiones generadas por el diseño hardware a 0b000. Valores booleanos indicando si alguna de las siguientes características debe aplicarse:

124 *No snoop*. Para una petición donde este flag está habilitado, ésta puede ser dirigida directamente hacia el controlador DRAM para la lectura/escritura de datos. Sin embargo, no se aplica ningún mecanismo de coherencia de cachés, por lo que el dato leído podría no ser el que actualmente está manejando el procesador o, en caso de escritura, no se actualizaría la información escrita (en caso de uso por la CPU) de manera inmediata.

125 *Relaxed ordering bit*. Las peticiones pueden ser respondidas en un orden no estrictamente coincidente al solicitado.

126 *ID-Based ordering bit*. Las peticiones pueden ser intercaladas temporalmente asegurando el orden creciente (en dirección) de cada identificador del solicitante.

127 *Force ECRC*. Insertar un código de redundancia cíclica de modo que se permita asegurar la detección de alteraciones en el paquete.

- Paquetes de petición originados en el hardware, Cuadro 4.3 (notar que campos a 'R' indican la carencia de sentido en ese tipo particular). En los bits [120:104] aparecen ciertas diferencias dependiendo del origen de la petición. En el caso de que el origen esté sustentado en una aplicación ejecutada en software se ha mostrado en el punto anterior. Sin embargo, en el caso de que las peticiones sean generadas en hardware se interpreta la información con el siguiente significado:

119:104 *Completer ID*. Identificación del destino de la solicitud. Por defecto, se utiliza la notación BDF pero si ARI está habilitado *device* y *function* son combinados.

120 *Requester ID enable*. Indica si el campo *Completer ID* presenta unos valores válidos o deben usarse los valores de la previa petición recogidas por el IP core de Xilinx.

- Paquetes de finalización a una petición generada por el software Cuadro 4.2. La respuesta es generada por el hardware reconfigurable en este caso. Muchos de los campos son compartidos con un paquete de solicitud, por lo que la descripción de los mismos es análoga a la del caso anterior y por simplicidad es obviada.

6:0 *Lower Address*. Para la respuesta a peticiones de memoria, este campo debe establecerse a los 7 bits menos significativos del bloque de memoria que se tiene que transferir. Para el resto de paquetes, estos bits deben figurar a 0. En esta implementación particular, siempre se trata del primer caso de la condicional.

9:8 *Address Type*. Para paquetes de finalización asociados a operaciones de memoria y operaciones atómicas este atributo debe presentar el mismo contenido que el paquete de petición. A 0 en cualquier otra situación.

28:16 *Byte count*. Se distinguen dos casos diferenciados:

* Si el tamaño de la petición se completa en un único paquete, se indica el tamaño del payload total en bytes. Cobra valores en el rango $[0, 4096]$.
* Si el tamaño de la petición no se completa en un único paquete, expresa el valor de bytes restantes para completar la operación. Es decir, indica el valor mostrado en el paquete previo menos el número total de bytes transferidos en el actual.

29 *Locked Read Completion*. Activado cuando se trata de la respuesta a una operación *Locked Read Request*.

42:32 *DWORD count*. Tamaño en palabras de 4 bytes del payload.

45:43 *Completion status*. Indican el posible estado de la operación:

000 Respuesta exitosa.

001 Petición no soportada.

100 Operación abortada por el *completer*.

46 *Poisoned completion*. Bit por defecto a 0 salvo que la aplicación de usuario haya detectado un error en los datos recibidos.

63:48 *Requester ID*.

71:64 *Tag* asociado con la petición.

87:72 *Completer ID*.

88 *Completer ID Enable*. Indica si el *endpoint* debe utilizar el campo *completer ID* o debe emplear el asociado al paquete de petición recogido por el IP core de Xilinx.

91:89 *Transaction class*. Misma interpretación que en los paquetes de la clase *request*.

94:92 *Attributes*. Misma interpretación que en los paquetes de la clase *request*.

95 *Force ECRC*. Obliga al *endpoint* a generar un código de redundancia cíclica sobre el contenido del mensaje.

- Paquetes de finalización a una petición generada en hardware, Cuadro 4.4. Se diferencia del caso de las peticiones originadas en software en las siguientes ubicaciones:

11:0 *Lower Address*. Se considera un total de 12 bits en lugar de los 7 iniciales.

15:12 *Error code*. Distintas anomalías pueden ser detectadas en la interpretación de un paquete:

0000 Terminación normal.

0001 El paquete TLP está envenenado (la información debe ser descartada).

0010 La petición ha sido terminada por un paquete de respuesta con código *completion status* no exitoso.

0011 Petición terminada por una respuesta sin datos (o por la recepción de un número mayor de los esperados).

0100 La respuesta tiene un tag que está asociado a otro elemento (ID, TC o atributos no corresponden).

0101 Error en la dirección de inicio. La dirección de inicio del paquete no corresponde con la siguiente esperable para ese flujo de datos.

0110 Tag inválido. Activado si se recibe una respuesta a un tag que nunca ha sido solicitado.

1001 Petición terminada por inactividad durante un determinado periodo de tiempo.

1000 Petición terminada por un reinicio de la función que aportaba dicha funcionalidad.

30 *Request completed*. Indica si el actual paquete es el último que integra la respuesta una petición.

45:43 *Completion status*. Indican el posible estado de la operación. Adicionalmente a los ya citados se añade el tipo 0b010:

000 Respuesta exitosa.

001 Petición no soportada.

010 *Configuration Request Retry Status*. Se ha realizado una solicitud pero el dispositivo aún no está habilitado para su funcionamiento. Por ejemplo, no ha terminado de inicializarse tras un reinicio.

100 Operación abortada por el *completer*.

B Publicación en ReConFig'15

El artículo ha sido enviado al *International Conference on ReConFigurable Computing and FPGAs*.

A PCIe DMA engine to support the virtualization of 40 Gbps FPGA-accelerated network appliances

Jose Fernando Zazo*, Sergio Lopez-Buedo*†, Yury Audzevich‡, Andrew W. Moore‡
*NAUDIT HPCN
Calle Faraday 7, 28049 Madrid, Spain
†High-Performance Computing and Networking Research Group, Universidad Autonoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain
‡Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom

***Abstract*—Network Function Virtualization (NFV) allows creating specialized network appliances out of general-purpose computing equipment (servers, storage, and switches). In this paper we present a PCIe DMA engine that allows boosting the performance of virtual network appliances by using FPGA accelerators. Two key technologies are demonstrated, SR-IOV and PCI Passthrough. Using these two technologies, a single FPGA board can accelerate several virtual software appliances. The final goal is, in an NFV scenario, to substitute conventional Ethernet NICs by networking FPGA boards (such as NetFPGA SUME). The advantage of this approach is that FPGAs can very efficiently implement many networking tasks, thus boosting the performance of virtual networking appliances. The SR-IOV capable PCIe DMA engine presented in this work, as well as its associated driver, are key elements in achieving this goal of using FPGA networking boards instead of conventional NICs. Both DMA engine and driver are open source, and target the Xilinx 7-Series and UltraScale PCIe Gen3 endpoint. The design has been tested on a NetFPGA SUME board, offering transfer rates reaching 50 Gb/s for bulk transmissions. By taking advantage of SR-IOV and PCI Passthrough technologies, our DMA engine provides transfers rate well above 40 Gb/s for data transmissions from the FPGA to a virtual machine. We have also identified the bottlenecks in the use of virtualized FPGA accelerators caused by the IOMMU. Finally, the DMA engine presented in this paper is a very compact design, using just 2% of a Xilinx Virtex-7 XC7VX690T device.**

***Index Terms*—Network Function Virtualization, Virtual Network Appliance, FPGA-based acceleration, SR-IOV, PCI Passthrough, PCIe, DMA engine, NetFPGA SUME**

I. Introduction

There is at the moment a huge interest for Network Function Virtualization (NFV) technologies. The advantages of moving from proprietary hardware appliances to virtualized software appliances are well known: Reduced costs and power, reduced time to market and service deployment, possibility of using a single platform for different applications and/or users, and, of course, it encourages openness and innovation [1]. Additionally, NFV complements very well Software-Defined Networking (SDN). Firstly, an optimal flexibility is achieved when everything is software-based (a software-based control plane manages software-based packet processing appliances). Secondly, the virtualization infrastructure can be used to provide a virtual machine where to execute the control plane.

The goal of NFV is to substitute proprietary, ASIC-based network appliances by software programs running on a virtual machine. Although ASICs definitely provide the best performance, there have been in the recent years a number of developments that allow removing the bottlenecks in a configuration based on a commodity Network Interface Card (NIC) and an off-the-shelf server. Specifically, libraries and drivers for fast packet processing such as DPDK [2] allow bypassing the burdensome networking stack of the operating system. Moreover, technologies such as PCI passthrough and SR-IOV allow improving the performance of virtual machines so that the overhead of virtualization is minimized [3].

The advantage of FPGAs is that they, to a certain extent, provide the speed of hardware and the flexibility of software. That is, one can take advantage of ASIC-like hardware acceleration without losing the reprogrammability of software. This is very interesting in an NFV environment: One could think of having FPGA-accelerated virtual networking appliances, composed by a software program and a FPGA bitstream. In this scenario, conventional NICs would be substituted in the servers by commodity FPGA boards with networking ports (such as NetFPGA). Therefore, a virtual network appliance would be deployed by launching a virtual machine in the server and reconfiguring the FPGA in the networking board. FPGA reconfiguration could be total or partial; the latter in the case where several virtual network appliances share the same board.

One of the difficulties of this FPGA-accelerated approach for virtual networking appliances is the communication between the software running on a virtual machine and the FPGA design. Fortunately, the same techniques being developed for conventional NICs apply in this case: PCI passthrough and SR-IOV. However, implementing in FPGA a PCIe DMA engine with these capabilities is far from trivial. In this paper, we present an open-source solution capable of achieving data transfer rates well above 40 Gbps for a FPGA to virtual machine transmission. These transfer rates are achieved by using PCI passthrough and SR-IOV as well as a huge-page, zero copy driver. This development is part of the NetFPGA SUME project.

II. Usage model

As we have already introduced, the big advantage of NFV technology is that a general-purpose cloud of servers, storage and Ethernet switches can be used to deploy a custom network that includes many specialized appliances (routers, firewalls, load balancers, etc.). In this paper, we envision adding commodity FPGA boards to the servers in order to improve the packet processing capabilities of virtualized software appliances. The FPGA boards will also feature network interfaces, so these boards will eventually replace the conventional Ethernet NICs of servers, or at least, complement them.

Therefore, virtual network functions will be composed of a software program and an FPGA design. Packets will arrive to the FPGA, where they will be processed. Processed packets will be either forwarded to other network interface in the FPGA board, or sent to the CPU for further processing in software. Therefore, this model allows for a flexible computing model, where network applications can be completely software, completely hardware or a mix of both. Fig. 1 shows an example, where two virtual FPGA-accelerated firewalls are running in one server. The FPGA board has 4 interfaces, each pair of them are dedicated to one virtual firewall. The FPGA device is split into two virtual accelerators, each running one instance of the hardware accelerator for the firewall. On the other side, the CPU runs two virtual machines, each running one instance of the software program for the firewall appliance. Communication between the software running in the virtual machines and the virtual hardware accelerators will be made via the PCIe bus of the commodity FPGA board.

A major challenge in this approach is how to establish an efficient communication between virtual machines and virtual hardware accelerators, avoiding contentions in the single PCIe bus and also avoiding any overheads related to the execution of software in a virtual machine. As it was mentioned in the introduction, two key technologies allow designers to overcome this challenge: SR-IOV and PCI passthrough. SR-IOV enables the creation of several virtual functions in the FPGA, each associated to one virtual machine in the CPU. PCI passthrough allows for a direct access of the virtual machine to the PCI bus, without the intervention of the host operating system. Using these technologies, the logical model for the two virtual FPGA-accelerated firewalls is that of Fig. 2. The software running on the virtual machine sees a direct connection to its virtual FPGA accelerator, without any interference from the host operating system or the other virtual appliance.

III. Related Work

References [4] and [5] provide a comprehensive summary of hardware acceleration techniques for NFV applications. However, the approach proposed in these papers differs from the tightly coupled FPGA accelerator model considered in this work. On the one hand, Kachris et al. propose in [5] an FPGA-based NFV platform where network functions are completely implemented in FPGA. They present a mechanism for different hardware accelerators to interact in an FPGA-based network appliance, and they also identify two separate interfaces to the SDN control and NFV controller entities. On the other hand, Nobach et al. in [4] present Elastic AH, an approach based on pools of software virtualized network functions (VNFs) and acceleration hardware modules (AHs). A network function is delivered as package composed of main software for VNFs and offloading programs for AHs. An OpenFlow switch directs traffic to either the VNF or the AH according to rules specific for each network function. For example, in an IPSec endpoint authentication is managed by the VNF, while payload encryption is performed by the AH.

A similar model to that followed in this work (tightly coupled FPGA accelerators) is presented in [6]. Although it makes reference to the use of PCIe SR-IOV technology, no implementation details are provided. Certainly, references are scarce in the NFV domain. However, more literature is available in algorithm acceleration field. [7] details how to communicate an FPGA accelerator with a virtual machine using PCI passthrough technology, although this work does not consider multiple functions in the FPGA. Nevertheless, other works in this algorithm acceleration field show how to split a single FPGA into several virtual functions. For example, [8] divides the FPGA device into several areas, each dedicated to implement a virtual FPGA accelerator. Partial runtime reconfiguration is used to change one virtual coprocessor without disturbing the others. [9] follows a complementary approach. Several virtual processing units are created out of a single FPGA board by using a time-multiplexed scheme.

Finally, [10] shows how to integrate FPGA accelerators in a cloud for NFV. Although this paper follows the model of isolated FPGA-based functions, the proposed methodology could be easily ported to the tightly-coupled accelerator scheme being considered in our work.

IV. Design overview

In order to provide a complete FPGA-accelerated NFV platform, both hardware and software components need to be implemented. The software components are basically the driver and the libraries that implement the API utilized by the higher-level user applications to access the FPGA accelerator. Of course, the kernel modules that composed the driver should be state-aware of the virtualization condition.

A key concept related to the operating system (OS) is the hypervisor or Virtual Machine Manager (VMM). Typically, the VMM plays a leading role in the arbitration process when VMs are involved. Different kinds of VMMs are well-known and can present different architectures [11]. From hypervisors integrated in the OS kernel to standalone solutions, the design proposed in this paper is intended to be run under all the different technologies, that is to say, there is no inconvenient in choosing one particular VMM. This is mainly thanks to the SR-IOV technique. SR-IOV inherits direct I/O technology by using an IOMMU to prevent the system from protecting memory accesses and translations. The penalty of setting up a system based on this kind of virtualization, will be drastically

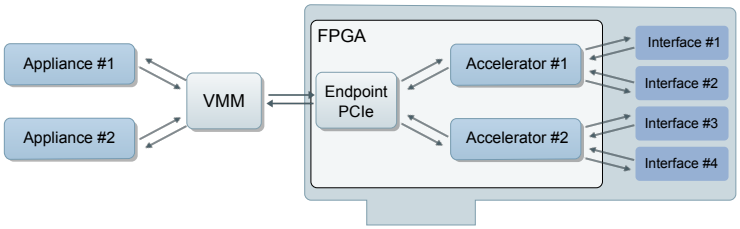


Fig. 1: Example of FPGA-accelerated firewall: Physical connections

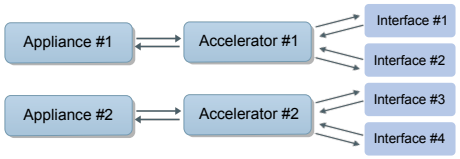


Fig. 2: Example of FPGA-accelerated firewall: Logical connections

reduced given the fact that no emulation of the device is required. In essence, multiple virtual devices, also known as Virtual Functions (VFs), are associated to a Physical Function (PF). A PF is a full-featured PCIe function, whereas a VF lacks configuration resources. A VF is meant to be plugged directly to a VM ensuring that a misconfiguration of the device cannot be done.

On the other hand, the hardware design is in charge of emulating and managing the several virtual devices. With a higher limit of 2 PF and 6 VF (imposed by the PCIe end point), the main goal is to keep a constant bandwidth with the best achievable performance among all the devices. By no means it can be omitted that BIOS SR-IOV support is required as long as the OS and hypervisor have to provide it too. Next section describes the key concepts of the design: DMA core, device drivers (virtual and physical drivers), user space programs and experimental test.

A. *DMA core*

Xilinx 7 Series Integrated Block for PCI Express [12] abstracts the process of reconstructing a packet from the physical PCIe lanes. Identifying and generating Transaction Layer Packets (TLPs) is the task that third components should carry on. First of all, some further considerations are required. There is no need in developing a logic that exactly corresponds with the whole PCIe specification. There are certain TLP types that are not worth useful for CPU-FPGA communications. According to the general classification two main types are distinguished:

- Non posted transactions are the ones where the requester expects a completion TLP from the completer side.
- Posted transactions does not receive a completion TLP after the end of the operation.

Classical examples are memory writes for the first group and memory reads for the posted group. Communication with the host is evaluated so no IO reads or IO writes are required in this design. According to this simplification, encapsulated by the end point, there are a total of four AXI4-Stream interfaces that need to be taken into account:

- Completer request (CQ). TLPs related with non-posted operations (memory reads from the FPGA to the Host memory) and posted operations (memory writes to the FPGA) are transmitted over CQ interface. The petitions come from a third agent: the CPU-based software.
- Completer completion (CC). Completion TLPs for CQ posted requests are propagated by this interface. This completions are created by the FPGA.
- Requester request (RQ). TLPs related with non-posted operations (memory reads from the Host memory to FPGA) and posted operations (memory writes to the host memory) are transmitted over RQ interface. The requests from this logic are generated by the own core instead of an external agent.
- Requester completion (RC). Completion TLPs for RQ posted requests are propagated by this interface. Its source is the PCIe root.

Basically, there is a main division between the source of the TLPs: completer and requester. CQ interface will be used for the initial configuration of the DMA in cases where an humongous quantity of data is desired to be moved. Whether this amount is reduced, the copy/read is feasible of being done by CPU direct access. They are the main purposes of the completer. Whereas, requester will implement the logic for copying huge regions of memory with the least intervention from the software.

Finally, but by no means least, MSI-x interrupts offer the capability of freeing the CPU from doing and unnecessary task (polling approach) whilst the hardware design is busy. An IRQ is supposed to be generated when a DMA successfully ends or,

in the case, where an abnormal execution is detected. MSI-x can manage up to a total of 2048 different IRQs, a wide variety that does not limit the number of concurrent VM in used by the architecture.

Two key concepts need to be explained in advance to the description of a DMA operation: engine and descriptor:

1) A descriptor is the piece of information with the basic fields that encompass all the necessary data for finishing an operation. A descriptor involves the address where the data will be copied/read, size in bytes of such operation and a boolean value indicating if an IRQ will be generated on completion at least. Status flags such as the current state, the active time or the number of bytes previously treated by this particular entity are also available. In this project, a descriptor is allocated in the hardware resources so they can be accessed by the CQ interface.
2) An engine is an unidirectional logic integrated by a group of descriptors with a topology of a circular list and which can operate in the C2S (card to system) or S2C (system to card) direction (mutually exclusive). The possibility of transferring a region of non-consecutive memory with an unique configuration by the CPU is called scatter-gather. This phenomenon is owing to the fact that the user specifies a pointer to the last descriptor in the list so the hardware design will be stopped by the time it is reached but will have consumed all the previous elements in the group of descriptors. Finally, each engine has a FIFO where the ordered information is consumed/stored by the core.

According to the specification of an engine, for a C2S operation the CPU-based design configures as many descriptors as possible in an engine which supports the C2S capability. Once it is done, the enable bit is asserted and the operation can now begin. Following this action, the information will be splitted according to the maximum payload (generally 64-256 bytes) and will be preceded by the correspondent header. No feedback from the host is expected so this procedure is iterated until having sent all the information. Another aspect that has to be taken into account is the PCIe credits. The PCIe endpoint limits the number of TLPs that can be transmitted at the same time. The number of tokens indicate the maximum number of TLPs that concurrently may be transmitted.

When the software logic configures a S2C operation, a TLP indicating the proper request type has to be formed. The asked region is limited in size to the maximum read request (128-4096 bytes) and many regions can be simultaneously asked. The window size is parameterizable and presents the main inconvenient that the completion TLPs might arrive unsorted. Here the tag field plays a decisive factor.

There is a FIFO associated to the each tag contemplated (greater numbers of the tag will be modulus by this number of maximum tags under observation), so when a completion arrives, the payload is stored in the correspondent FIFO. By the time when as many words as were requested have arrived, this tag can be used again and the information can be delivered to the rest of the hardware logic.

B. SR-IOV

SR-IOV is supported by the Integrated Block for PCI Express so an according configuration has to be adopted. In this design a total of 1 PF and 1 VF are created. Enabling DMA transfers is straightforward once the DMA core is developed.

The routing of the TLPs is now carried by the BDF notation, Bus number:Device number:Function number (8 bits/5 bits/3 bits). The bus number has to be remained intact so a total of 8 bits are available. The PF will be represented by the values 0x00 and 0x01 in the pair Device-Function. VF functions by the value from 0x40 to 0x47. This way the hardware design can clearly distinguish the source/destination of a TLP in the completer interfaces. In the requester interfaces there is no such feature so the tag is used as the device identifier. The tag has a width of 8 bits, so a total of 4 bits are used for redirecting between the PF and VF whereas the rest are used in order to sort the information in a S2C operation.

The general structure is shown in Fig. 3 where as many components as needed of the previous cores are instantiated. The completer side suffers no variations but taking into consideration the completer id so the proper virtual device can be inferred.

C. Physical and Virtual driver

Physical and virtual driver are practically analogous in behaviour. Nevertheless, the physical driver, in exclusive, access the PF resources, enables the VFs and sets the maximum payloads and read requests. Both drivers can attend IRQs and support CPU direct accesses to the device and DMA operations. The general steps for communicating with the FPGA are following stated:

1) The user notice the necessity of read/write data from/to the hardware design.
2) The user can do it directly consuming CPU time or rely this activity on the SRIOV core.
   - If the DMA-SRIOV core is not involved, the affected interfaces are CQ and CC. This AXI4-Stream interfaces are treated and converted into a memory-like interface. If the destination address does not coincide with the reserved areas for configuring the DMA-SRIOV core, the information will be provided to the user hardware application eventually.
   - If DMA operations are desirable, prior to any transference the core has to be configured (one engine and at least one descriptor). For this purpose, CQ and CC interfaces are used and once the configuration is done, the AXI4-Stream interfaces RQ, RC will be involved.
3) The design is stopped until the end of operation or more descriptors can be queued for its concurrent transmission. It must be taken into account that the user is not allowed to alter the information of a descriptor that has not been processed yet. The device driver is

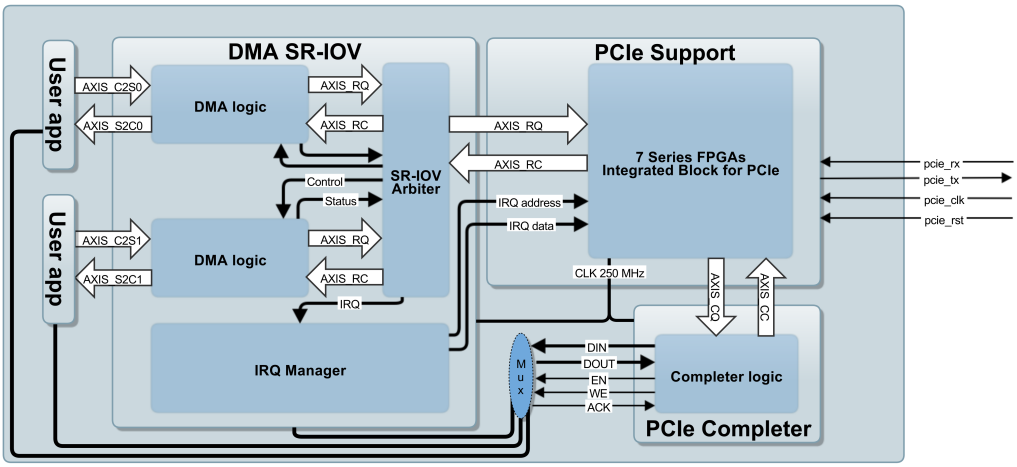


Fig. 3: Block diagram of the hardware architecture

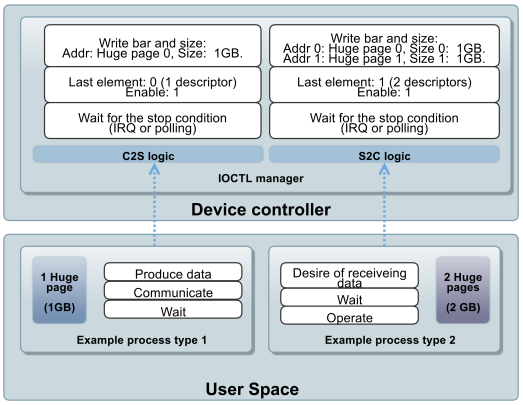


Fig. 4: User space example procedure

aware of the current descriptors that are in used by the hardware design and it updates the last element pointer accordingly if more data is needed.

Every VF has its own DMA logic, Fig. 5. That is to say: proper IO data FIFOs, a MSI-x table vector and pending bit array (PBA) in addition to a ring of descriptors for every engine (by default one S2C and one C2S engine). When a piece of information is desired to be delivered, ideally as many descriptors as non-contiguous regions of memory will be configured. It involves to supply address (64 bits), size (64 bits) and a boolean value indicating if an IRQ is desired to be generated by the end of the operation. On top of that, the number of involved descriptors and a enable bit will have to be properly asserted by the time the transference is supposed to start. In the Fig. 4 this process is detailed omitting the VM layer.

Finally, huge pages are highly advisable in order to allow bigger regions to be transferred in an unique descriptor at the same time that they affect positively in the VM general performance (whole system running over pages of a size bigger than 4KB and contiguous in main memory).

D. Experimental test

The developed prototype for S2C direction will fill a buffer sustained in huge pages with integers. The hardware design will check that those integers are in increasing order and that there are no gaps between two consecutive valid pieces of data. The experiment is reversed for the C2S direction: the hardware will generate the integers whilst the user space application will check their correct value.

One VM and one VF are created using the hypervisor KVM. The main reason for choosing KVM is the stability of the platform and the free license that accompanies the software. Additionally, most current Linux distributions offer support for this kernel module so its implementation in other environments could be less harmful to the original software configuration. For the experiment, the VM will have available 8GB of RAM memory and a maximum of 4 CPU cores. VT-x, VT-d and SR-IOV capabilities are enabled at BIOS and host OS level.

The chosen architecture for the experiment consists on a Supermicro X9DRD-iF motherboard, two Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz microprocessors, 64 GiB of DDR3 RAM at 1600 MHz (8 banks of 8 GiB, 4 modules connected to each CPU) and a NetFPGA SUME board (featuring a Virtex-7 XC7VX690T-2FFG1761C FPGA) directly attached to the first CPU. Thread affinity is treated in order that all the computing tasks are mapped to the first CPU in order to totally isolate the second CPU of data interchange operations.

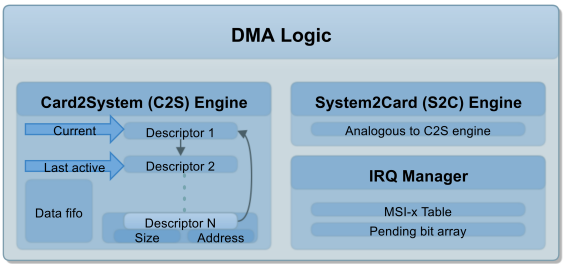


Fig. 5: DMA logic implementation at hardware level

Slice logic	DMA Design	SR-IOV Design
FF	8,784 (1,03%)	8,951 (22%)
LUTs	9,171 (2.13%)	9,238 (2.13%)
Memory LUTs	136 (0.08%)	136 (0.08%)
Block RAMs	19 (1.29%)	19 (1.29%)
BUFGs	5 (15.62%)	5 (15.62%)

TABLE I: Device Utilization Summary

V. Results

A. Design occupation

The reference board is NetFPGA SUME. In Table I the device utilization is summed up.

B. Theoretical throughput results. PCIe rate

PCIe 3.0 offers the bandwidth of 8 Gbps per lane. In this case 8 lanes are considered so a theoretical rate of 64 Gbps could be achievable. However, due to the physical codification (128/130b) and the overhead of TLPs headers this rate is reduced. A supremum for the C2S direction, based on the fact that an encapsulated TLP frame overhead is 192 bits in this case (4 bytes from the physical layer, a 4-byte LCRC and a 16-byte TLP header given the use of 64 bit address), is given by the expression:

$$64 * 10^9 \frac{b}{s} * \frac{128}{130} * \frac{MAX\_PAYLOAD * 8}{MAX\_PAYLOAD * 8 + 192}$$

It is translated into a maximum achievable rate of 57.61 Gbps in the case where the maximum payload is 256B and 53.06 Gbps when the payload is 128 bytes. The S2C direction will be typically more critical in the sense that TLPs has to be built after a petition to the completer is sent.

C. Empirical throughput results

In Fig. 6 and Fig. 7 the results of the experiment are plotted. Four are the basic cases: DMA transactions with a hardware design where no SR-IOV support is offered (native), DMA transactions over the PF, DMA transactions over the VF (attached to the host machine) and DMA transactions over the VF (attached to a VM).

The general tendency is clear: small quantities of data are not worth being transferred in a DMA operation. DMA can be considered where the amount of data is less than

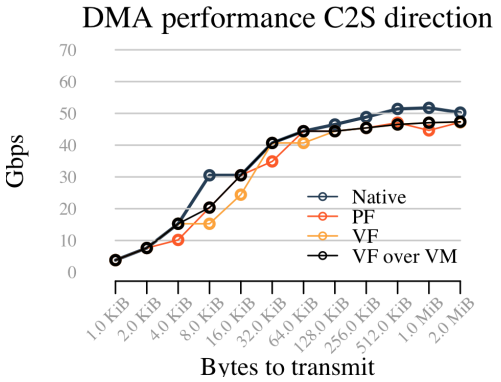


Fig. 6: Performance for C2S transferences

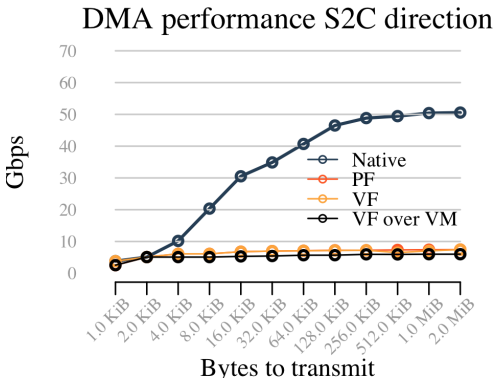


Fig. 7: Performance for S2C transferences

256KB approximately. The throughput tends to stabilize after this value offering a fulfilling result. In the C2S direction a maximum of 51.74 Gbps is reached when copying a region of 1 MB and in the S2C, a total rate of 50.40 Gbps is observed. Configuration of descriptors has to be made in advance so the theoretical rate would not be able to achieve in any case as long as this time is considered in the measure.

If attention is payed to the C2S direction, the behaviour is pretty similar in every technology. Transmissions over 128 KB assures at least the 90% of the native performance. However, if multiple VMs are trying to communicate data at the same time, their petitions are queued so the process of delivering the information may be delayed.

In the S2C direction the results are disappointing. From an initial value of 50.40 Gbps in a transmission of 1 MB in the native scene, a VM using the VF would be only able to achieve a maximum of 6.00 Gbps (11.90% of the native case). The justification resides on the fact that SR-IOV limits the virtual devices to present a maximum read request and a maximum payload of 128 bytes. It involves that C2S operations have decreased (original maximum payload of 256 bytes) but not

so notoriously as the S2C transfers (maximum read request of 4096 bytes). This limitation imposes a huge overhead in the movement of data from the system to the FPGA.

VI. Conclusions

In this paper we have presented the design of a PCIe DMA engine with full support for virtualization (SR-IOV) and capable of transferring data from the FPGA to the host at rates higher than 40 Gbps. This block is the key element to enable the FPGA acceleration of virtual network appliances. The final goal of this work is to have an NFV architecture where conventional NICs are substituted by FPGA boards, in order to overcome the limitations at very high rates (40+ Gb/s) of software-only implementations.

The benefit of the proposed design is that it is a fully scalable block, capable of creating several virtual functions, each with one or more DMA engines. As a result, very complex functions can be created in the FPGA. We have shown that the overhead of implementing SR-IOV virtualization for the card to system transfers is minimal, and we have also identified the bottleneck caused by the IOMMU in the system to card direction. Additionally, the design is very compact, occupying just 2% of the selected device for the minimal configuration.

This development is part of the NetFPGA SUME project and, as an open-source design, the goal is to foster research in FPGA-accelerated NFV. As future work we envision the integration of this DMA engine with a framework for the dynamic reconfiguration of the FPGA, in order to support several virtual accelerators in one FPGA, that can be changed at runtime. This way, the virtualization of networking appliances would be complete: Both at the software and at the hardware acceleration levels.

Acknowledgment

This work was partially supported by the Spanish Ministry of Economy and Competitiveness under the project PackTrack (TEC2012-33754) and by the European Union through the Integrated Project (IP) IDEALIST under grant agreement FP7-317999.

References

[1] European Telecom Standards Institute (ETSI), "Network functions virtualisation introductory white paper," October 2012.

[2] Intel. Intel DPDK: Data plane development kit. [Online]. Available: http://dpdk.org/

[3] IBM. Linux virtualization and pci passthrough. [Online]. Available: http://www.ibm.com/developerworks/library/l-pci-passthrough/

[4] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in nfv environments," in *Networked Systems (NetSys), 2015 International Conference and Workshops on*, March 2015, pp. 1–5.

[5] C. Kachris, G. Sirakoulis, and D. Soudris, "Network function virtualization based on fpgas: A framework for all-programmable network devices," June 2014. [Online]. Available: http://arxiv.org/abs/1406.0309

[6] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "Openanfv: Accelerating network function virtualization with a consolidated framework in openstack," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 353–354. [Online]. Available: http://doi.acm.org/10.1145/2619239.2631426

[7] W. Wang, M. Bolic, and J. Parri, "pvfpga: Accessing an fpga-based hardware accelerator in a paravirtualized environment," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, Sept 2013, pp. 1–9.

[8] E. El-Araby, I. Gonzalez, and T. El-Ghazawi, "Virtualizing and sharing reconfigurable resources in high-performance reconfigurable computing systems," in *High-Performance Reconfigurable Computing Technology and Applications, 2008. HPRCTA 2008. Second International Workshop on*, Nov 2008, pp. 1–8.

[9] I. Gonzalez, S. Lopez-Buedo, G. Sutter, D. Sanchez-Roman, F. J. Gomez-Arribas, and J. Aracil, "Virtualization of reconfigurable coprocessors in hprc systems with multicore architecture," *J. Syst. Archit.*, vol. 58, no. 6-7, pp. 247–256, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2012.03.002

[10] S. Byma, J. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 109–116.

[11] M. Tim Jones, "Anatomy of a Linux hypervisor," May 2009. [Online]. Available: http://www.ibm.com/developerworks/library/l-hypervisor/

[12] Xilinx, "Virtex-7 fpga gen3 integrated block for pci express v4.0 logicore ip product guide 023, vivado design suite," July 2015.